

# The Fundamentals:

*Building Visual Studio Applications  
on a Visual FoxPro 6.0 Foundation*



*Whit  
Hentzen*

*Edited by  
Doug Henning*

**Hentzenwerke Publishing**



## **Dedication**

*This book is dedicated to the memory of Tom Rettig.*

*Tom epitomized the spirit of “Proper Programming Practices,”  
and to this day, I have always used him as the ultimate measure of my code:  
“What would Tom think of this?”*

*You are missed, Tom.*



# Acknowledgements

You might think that the coolest part of writing a book on a piece of software is getting to play with new software before the general public sees it. I hate to burst your bubble, but actually, the best part is getting to meet new people and work with old friends during the project.

First comes my tech editor and good friend Doug Hennig. (He's also an old friend, although less old than me by about a month.) If I had to pick one person in our community who could rightfully assume the position of Master Developer left by Tom Rettig, it would be Doug. I was damned lucky that his wonderful bride, Peggy, and his adorable little boy, Nick, were generous enough to give up some of their time with him while he was editing this tome.

Next is my superb copy editor and layout pro, Jeana. We first met back in the mid-'90s (I think) when I was scribbling an occasional article for *FoxTalk*. She's worked on each of the Essentials books so far, and we've all been very fortunate to have her by our side.

Thirdly, I want to thank all the readers of my *Programming Visual FoxPro 3.0* book, and all the patient customers of this book. You've been very nice to have waited this long; I hope this book pleases, and that you will tell me what you like—and don't like! I've learned my lesson: I won't try to start another company and write two other books while I revise this book for version 7.0!

Where to begin with the Fox community? I started listing names, but felt I was missing people, so I opened my Outlook and CompuServe address books and found that I've accumulated more than 2,000 names over the years. So I know that I will miss a few—probably many.

There are the authors and tech editors of the books in the Essentials series: Jim, Steve, Steve, Ted, Tamar, Steven, Drew, Markus, Mac, Rick, Gary, and Jeff. And those working on books for the future—Della, Andy, Marcia, John, Mike, and Chaim—or who contributed in one way or another: PAul, Rick, Kris, Bill, Barbara, Andrew, Jim, and Patty.

There are the folks on the VFP product team—both past and present—Alison, Susan, Randy, Ken, Calvin, Robert, Winnie, Ricardo, John, Imtiaz, Hong-Chee, John, Mike, Jim, Gene, Garrett, Mike, Doug, Steve, Tom, and McAlister.

And the UT folk, including Michel, John, Barb, Nancy, Ed, Rox, Dave, Craig, Arnon, George, Vlad, and, of course, Lulu.

There are also folks responsible for, as they say, "atmosphere"—Val, Igor, Paul, Menachem, Jim, Dan (P.A.M.), Kelly and that bunch, Ceil, Colin, >L<, YAG, Eldor (The PEBCAK Man), Fletcher, Chuck, Mike, Toni, Phil, Mickey and The Dog, Ellen, wOOdy, Rainer, Volker, and the rest of the German contingent, jMM, JVP, Rod, Ed, Chick and Bill, Scott, Laura, Laura, Laura, the Silver Fox, Paul, Jacci, Mike, John and DTA the tag team, Art, Doug, Dick, and Heather.

And let me not forget Billy Preston!!

Finally, it's time for attributions of the bands that kept me awake in the wee hours this winter and spring while I was experimenting with this feature or that, or performing yet another NT install due to the vagaries of the beta-testing process. Although they'll never see this, nor would they care if they did, I must mention Blues Traveler, Gravity Kills, The Poor, Queensryche, Supertramp, and, proving that the truth isn't always very pretty, Rob Zombie.

Software is always better when written at 105 decibels.

Whil Hentzen  
Milwaukee, WI



# Boring Stuff About the Author

## Whil Hentzen

Whil Hentzen is president of Hentzenwerke Corporation, a 17-year-old firm that specializes in strategic database applications for Fortune 2000 firms in the manufacturing, financial, and health-care industries. The firm has commercial products and custom applications in use throughout the United States and in nearly two dozen foreign countries. Hentzenwerke has hosted the semi-annual Great Lakes Great Database Workshop since 1994.

Whil has written and spoken extensively about software development. He is a multi-year Microsoft MVP, and editor of *FoxTalk*, Pinnacle Publishing's high-end technical journal for FoxPro. He is the author of books about Visual Studio (*Visual FoxPro 6.0 Fundamentals*), Visual FoxPro (*Programming Visual FoxPro 3.0*), FoxPro (*Rapid Application Development with FoxPro 2.6*), and custom software development (*1999 Software Developer's Guide*).

He has presented more than 50 papers at conferences throughout North America and Europe, including the Microsoft Visual FoxPro DevCon, the German National DevCon, Conference to the Max (the Netherlands), the Spanish National DevCon, Database & Client/Server World, FoxTeach, the FoxPro Users Conference, and the Mid-Atlantic Database Workshop.

He spends his spare time with his kids and volunteering for the local school district, and is an avid distance runner, hoping for one more shot at a sub-15-minute 5,000-meter clocking before age and common sense close the door on that activity.

You can reach Whil at [whil@hentzenwerke.com](mailto:whil@hentzenwerke.com).

## Doug Hennig, Technical Editor

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He's the author of Stonefield's Database Toolkit for Visual FoxPro and Stonefield Data Dictionary for FoxPro 2.x. He's also the author of the *Visual FoxPro Data Dictionary* in Pinnacle Publishing's *The Pros Talk Visual FoxPro* series. Doug has spoken at user groups and regional conferences all over North America, and at the last three Microsoft Visual FoxPro Developer Conferences. He's a Microsoft Most Valuable Professional (MVP.)

You can reach Doug at [dhennig@stonefield.com](mailto:dhennig@stonefield.com)



# How To Download the Files

To download the files that accompany this book:

1. Point your web browser to [www.hentzenwerke.com](http://www.hentzenwerke.com)
2. Look for the link that says “Download Source Code & .CHM Files”
3. If you were issued a username and password from Hentzenwerke Publishing, enter them. Otherwise, enter your email alias and two pieces of data from the book(s) that you’re interested in. You’ll need the book itself when you do this.
4. You’ll get a page that lists hyperlinks to the files for the book(s) you’ve selected.

If you have questions or problems, please email [webmaster@hentzenwerke.com](mailto:webmaster@hentzenwerke.com).

Please note: The .CHM file is covered by the same copyright laws as the printed book. Reproduction and/or distribution of the .CHM file is against the law.



# Read This First!

When dBase IV shipped, it was noteworthy in several respects. One was the packaging: It was contained in a 20+ lb box—arguably the biggest product that had shipped in the PC world to date. In this box were oodles of manuals and about a billion diskettes. It was also notable that the box contained three separate pieces of paper that all demanded “Read This First!” And finally, it was notable because the product worked rather poorly, although this attribute has been adopted by a number of other software manufacturers since.

When you open most books, you see an “Introduction” or a “Prologue” or a “Before We Begin” or a “What You Need To Start Out.”

I’m not going to do that to you—this is where you should start. So **READ THIS FIRST!**

## Purpose

The purpose of this book is to get you, an intelligent computer user who has the wisdom and foresight to choose Visual FoxPro as one of their development tools, up to speed with Visual FoxPro 6.0. Whether you are new to the entire Fox environment or upgrading from a previous version of FoxPro—specifically FoxPro 2.6 for Windows or FoxPro 2.6 for DOS, but also Visual FoxPro 3.0—this book will take you from the very beginning and have you comfortable with building traditional LAN applications by the middle of the book.

But that’s not all! Yes, there’s more!

Visual FoxPro is no longer used only for traditional LAN applications—the things that sit on top of a Novell Server, dishing out records to 20 to 50 users in the same department or building. It’s also a fully capable, robust front end for client-server applications, where the data resides on a database server in a tool such as SQL Server. And it’s starting to be used as the glue between an indeterminate front end, such as a Web browser, and a similarly indeterminate back-end data store, such as Visual FoxPro tables or a SQL Server database.

So while the first half of this book gets you comfortable with the basic tools, there’s more under the hood. In the second half of the book, I’ll discuss the various tools that you’ll use to build advanced applications, both for the LAN as well as client-server and three-tier environments. Finally, I’ll wrap up with an introduction to several Visual Studio-level topics—concepts that transcend a single development environment.

But there’s more! Space and time limitations have prevented me from including complete client-server and three-tier applications in these pages, but the Fundamentals .CHM file that you can download along with the source code at [www.hentzenwerke.com](http://www.hentzenwerke.com) includes a couple of bonus chapters that port the sample LAN application to a client-server environment and to a three-tier structure with a Web-based front end.

## Icons and stuff

There are several icons used in this book. Here's what they mean:



Paragraphs marked with this icon indicate that the reference tool or application is available for download at [www.hentzenwerke.com](http://www.hentzenwerke.com). The source code for each chapter is in a .ZIP file with a name of format CHnn, where “nn” is the chapter number. All the code for a chapter is contained in that file—you don't have to unzip Chapters 1 through 22 in order to use the source code in Chapter 23.



Information of special interest, related topics, or important notes are indicated by the “note” icon.



Tips—marked with this icon—include ideas for shortcuts, alternate ways of accomplishing tasks that can make your life easier or save time, or techniques that aren't immediately obvious.

## What you need to work through this book

First off, you'll need a copy of Visual FoxPro 6.0. This book was written using VFP 6.0, Service Pack 3, running on Windows NT Workstation 4.0, SP 4 or 5 (I can't keep track of them all!). I would suggest that you install all of Visual Studio if you've got the disk space—there are tools and components included with VS that don't get installed if you just install VFP by itself.

As far as hardware, I started out with a 266 MHz PII with 64 MB of RAM, and that was a bit pokey at times. I'm currently using a 500 MHz PIII with 256 MB of RAM and a 21-inch monitor—and that's plenty of horsepower to run VFP. Regardless of what machine you use to develop on, be sure to keep one of those old P120 clunkers around so you can test your app on a machine that more closely mimics what your users will be running!

## A note about the source code

Both Doug and I have tested the source code for each chapter, so it should run for you. However, because it's demo code prepared specifically for this book, it has not been wrung out in an industrial fashion; you could very well find a bug or two. If you find a showstopper, let me know and I'll rework and repost the new source code. You can check to see if new code has been posted by looking at the “News” section of [www.hentzenwerke.com](http://www.hentzenwerke.com) periodically.

I'd also like to caution you that the source code included with this book is suited for your own application development. You can use it to get started, and you can borrow from it, but it is not intended to serve as your development framework!

# Table of Contents

<b>SECTION I—The Interactive Use of Visual FoxPro</b>	<b>1</b>
<hr/>	
<b>Chapter 1: Introduction to the Tool</b>	<b>3</b>
Today's computing environment	4
Installing and starting Visual FoxPro	5
The Visual FoxPro interface	5
The Command window	6
Windows, menus, and toolbars	8
Controls	8
Conclusion	11
<hr/>	
<b>Chapter 2: The Language</b>	<b>13</b>
Working with Visual FoxPro interactively	14
Using the Command window	14
The Command window context menu	15
Building commands, functions, and expressions	17
Commands	17
Variables	18
Operators	20
Functions	20
Expressions	21
A Visual FoxPro command and function overview	22
Arrays	23
Client/Server	24
Colors	25
Data – Database	26
Data – Editing	27
Data – Fields	28
Data – Indexes	28
Data – Locking	30
Data – Memo Fields	30
Data – Navigation	31
Data – SQL commands	31
Data – Structure	32
Data – Tables	32
DateTime	35
DDE	36
Debugging	37
Environment – General	37
Environment – Disk	38

<b>Environment – Keyboard</b>	38
<b>Environment – Monitor</b>	38
<b>Environment – Network</b>	38
<b>Error handling</b>	39
<b>File name manipulations</b>	39
<b>File information</b>	40
<b>File selection</b>	41
<b>File and directory writing</b>	41
<b>Fonts</b>	42
<b>Help</b>	42
<b>Interrupts (Mouse, Keyboard, Other)</b>	42
<b>International</b>	43
<b>Low-level file functions</b>	44
<b>Macros</b>	45
<b>Math</b>	45
<b>Math – Financial</b>	46
<b>Math – Numeric</b>	46
<b>Math – Statistical</b>	47
<b>Math – Trig and Exponential</b>	47
<b>Memory</b>	47
<b>Miscellaneous</b>	48
<b>OOP</b>	48
<b>OOP – Control-specific</b>	49
<b>Printing</b>	50
<b>Programming – Comments</b>	51
<b>Program control structures</b>	51
<b>Program event handling</b>	52
<b>Program subroutines</b>	52
<b>Program variable scoping</b>	53
<b>Project manager</b>	53
<b>String functions</b>	54
<b>Text merge</b>	56
<b>User interface</b>	56
<b>Variables</b>	57
<b>VFP</b>	58
<b>Visual Studio</b>	59
<b>Windows/menus</b>	60
<b>Obsolete commands and functions</b>	63

<b>Chapter 3: The Interactive Data Engine</b>	<b>69</b>
The relational model	70
Visual FoxPro's data structures	72
Tables	72
Fields	73
Nulls	75
Indexes	76
Original table	77
Index file (for order by Birth Date)	77
Table displayed according to Birth Date index	77
Index file	78
Table displayed according to Sex/Name tag	78
Indexes and Rushmore	79
Databases	79
.MEM files	80
Other FoxPro files	80
Working with data structures	81
Manipulating tables	81
Accessing an existing table	82
Exclusive vs. Shared use	83
Accessing multiple tables	84
Closing a table	86
Viewing the contents of a table	87
Specifying the order of records in a table	88
Creating indexes	88
Creating a table and modifying the structure of an existing table	91
Setting a filter	92
Manipulating databases	93
Accessing an existing database	93
Accessing multiple databases	94
Viewing the contents of a database	94
Using the Database Designer	96
Working with persistent relationships	101
Working with table structures programmatically	102
Modifying tables and databases programmatically	107
Working with data	109
Navigating through tables and finding specific data	109
SKIP	109
BOF(), EOF() and the Phantom Record	109
GO	112
SEEK	112
RUSHMORE—revisited	113
LOCATE	114
Modifying data	115
Adding records interactively	115

Adding records programmatically	115
Adding a batch of records from another source	117
Deleting records interactively	117
Deleting records programmatically	117
Handling deleted records: recalling and packing	117
Editing records interactively	119
Editing records programmatically	119
How do I save the data?	120
<b>Chapter 4: Visual FoxPro's SQL Implementation</b>	<b>123</b>
<b>SELECT</b>	<b>123</b>
Sample databases	123
Basic SELECT syntax	124
Field lists	124
Record subsets	126
Aggregate functions	129
Subtotaling—GROUP BY	130
Multi-Table SELECTs	130
Outer joins	131
Controlling the result set: destination	140
Controlling the result set: ORDER BY	141
Controlling the result set: DISTINCT	142
Controlling the result set: HAVING	143
<b>INSERT</b>	<b>143</b>
<b>DELETE</b>	<b>145</b>
<b>UPDATE</b>	<b>145</b>

## **SECTION II—Tools Needed for Building an Application** **147**

<b>Chapter 5: Creating Programs</b>	<b>149</b>
Creating a program with the editor	149
Running a program	151
Compiling a program	152
Up close with the VFP editor	152
Configuring Visual FoxPro program execution preferences	159

<b>Chapter 6: The VFP Project Manager</b>	<b>161</b>
Using the VFP Project Manager to build an .APP or .EXE	161
Up close with the Project Manager	162
The Project Manager dialog	162
The Project Manager context menu	165
The Project menu	169
Tips and tricks	169
<b>Chapter 7: Building Menus</b>	<b>173</b>
Driving an application with a menu	173
Using the Menu Designer to create a menu	174
Up close with the Menu Designer	176
The Menu Designer dialog	176
The Menu menu	180
The View menu	181
The Menu-generation process	184
Tips and tricks	185
Adding flexibility to the Visual FoxPro Menu Designer	187
Creating context menus	189
So why isn't the Menu Designer OOP-ified?	191
<b>Chapter 8: Building Forms</b>	<b>193</b>
The big picture	193
Terminology and tools	195
Creating and running a form	197
Changing form properties through user input	200
Calling methods from objects on the form	203
Integrating forms and data	206
Up close with the Form Designer	209
Form Designer	209
Properties window	210
Code window	214
Form Designer toolbar	214
Form Controls toolbar	214
Layout toolbar	216
Menus	216
Tab order	218
Include files	220
Referencing objects	221
Referencing objects by their full names	221
Referencing objects by their container	222
Indirect object references	223
Tips and tricks	223
Referencing generic objects	223
Names and captions	224

Code windows	224
Layout	225
<b>Chapter 9: Native Visual FoxPro Controls</b>	<b>227</b>
The proper care and feeding of controls	227
Focus and containers	227
Common properties	228
Common events	231
Event firing	232
Visual delineators	234
Label control	234
Image control	234
Line control	234
Shape control	234
Page Frame control	234
Separator control	235
Controls to initiate action	236
Command Button control	236
Command Button Group control	238
Timer control	238
Hyperlink control	240
Controls to manipulate data	240
Text Box control	240
Edit Box control	243
Check Box control	243
Option Group control	244
List Box and Combo Box controls	245
Uses of lists and combos	245
Populating a list or combo box	246
List Box functionality	250
Combo box functionality	250
Defining and returning values from a list box	251
Spinner control	251
Grid control	252
Other controls	254
Container	254
ActiveX control and ActiveX bound control	255
Custom class	255
Session class	255

<b>Chapter 10: Using VFP's Object- Oriented Tools</b>	<b>257</b>
A quick introduction to object-oriented programming	259
The original form and control objects	260
Inheriting from the original	260
Creating your own originals	260
OOP terminology	261
Inheritance and overriding methods	262
Non-visual classes	263
Where OOP fits in with Visual FoxPro	265
Quick start to creating classes	268
Form classes	268
Creating your own form base class	268
Using your own form base class	269
Creating a form from a Form Class template	270
Using a different form base class "on the fly"	271
Creating form subclasses	271
Registering a form class with Visual FoxPro	273
Attaching a custom icon to a registered class	274
Modifying a form class	275
Control classes	275
Creating your own control base class	275
Creating control subclasses from your base class	276
Registering your controls base class	276
Placing controls on a form from your base class or a subclass	277
Use the Project Manager	277
Select the class from the Form Control toolbar	277
Control and container classes	277
Working with Visual FoxPro's object orientation	279
Naming classes	279
Adding your own properties and methods	280
The property and method hierarchy	282
Overriding property inheritance	282
Overriding method inheritance	283
Preventing a parent's method from firing	283
Preventing an event from firing	284
Calling a parent's method in addition to the local method	284
Suggested base class modifications	285
Up close with the Class Designer	288
Up close with the Class Browser	289
Starting the Class Browser	289
Using the Class Browser	289
<b>Chapter 11: Output: The Report and Label Designers</b>	<b>293</b>
Quick start to building reports	293
Report terminology	293

To-may-to or tuh-mah-to?	294
Dumbing down the Report Writer	295
How about an example?	295
<b>Up close with the Report Designer</b>	<b>298</b>
The Report Designer window	298
Select a control	299
Select a group of controls	300
Move a control within a report	300
Move a control from one report to another	300
Duplicate a control	300
Resize a control	300
Delete a control	300
Change the font or other attributes of a control	300
Resize a band	300
The Report Controls toolbar	301
Change the text of a label	301
Change the internal properties of a control	301
Create a control	301
The Layout Controls toolbar	301
The Color Palette toolbar	302
The Report menu	302
The View menu	304
<b>Up close with the Label Designer</b>	<b>305</b>
The Label Designer window	305
<b>Tips and tricks</b>	<b>306</b>
Is that all there is?	307

## **SECTION III—Building Your First Application** 309

<b>Chapter 12: The Structure of a LAN Application</b>	<b>311</b>
Defining the application	313
Deciding on a menu style	313
Function-centric menu styles	313
CUA menu styles	314
Document-centric menu styles	315
A pseudo-object-oriented menu style	316
The Books and Software Inventory Control Application menu structure	317
Forms menu	317
Edit menu	317
Processes menu	318
Reports menu	318
Tools menu	318
Window menu	319
Help menu	319

Running a menu	321
Building a main program and an event handler to hold up the menu	322
Enhancing the main program	323
Starting the move to OOP	328
A quick review of non-visual classes	329
Creating a non-visual class	329
Implementing a non-visual class	335
Making setting up and cleaning up more automatic	339
Enhancing the non-visual class	339
Creating an application-level non-visual class	341
<b>Chapter 13: Building a LAN Application</b>	<b>347</b>
More on the sample application	348
Types of forms	349
Setting the stage: creating your class libraries	350
Creating HWCTRL62.VCX, your control class library	350
Create your controls class library	351
Register the library	354
Creating your form classes	354
Create your form base class	354
Create your maintenance form class	356
Set the form template class	358
Creating a real form from your base class	359
Create the ORDERS maintenance form	359
What's happening behind the scenes	362
Inheritance at work!	363
Thinking about more application modifications	363
<b>Chapter 14: Your First LAN Application: Building Upon Your Forms</b>	<b>365</b>
Enhancing your form class with toolbars and menus	365
Putting it all into a project	365
Creating a form with its own methods	365
Creating a new form based on the new form class	368
Adding a toolbar	372
Creating new control classes in HWCTRL62	373
Creating your base toolbar class	375
Getting your toolbar to work with the app	381
Consolidating code in a generic method	382
Getting rid of the command buttons	383
Creating an instance handler	385
Getting the toolbar to dock automatically	388
Adding the form name to the Window menu	388
Enhancing your form class with generic methods	389
An introduction to MessageBox() and #INCLUDE	390

Toolbar positions	391
MessageBox parameters	391
MsgBox return values	391
Cursor buffering modes	392
Data handling—buffering and multi-user issues	392
A simple error handler	393
Causing a conflict—pessimistic row buffering	394
Optimistic row buffering	395
A generic Save() method	396
A generic Add() method	401
A generic Delete() method	407
Adding hooks to generic methods	408
<b>Chapter 15: Your First LAN Application:</b>	
<b>Odds and Ends</b>	<b>411</b>
Adding more functionality to your customer form	411
The Sort Order combo box	411
Populating the combo box in the toolbar	412
Changing the order of the records	413
Concepts for child forms	414
Passing parameters to a form	414
Returning values from a form	415
Passing object references	415
Adding a list box to display child records	418
Populating a childbearing list box	418
Adding children	422
Editing an existing child record	425
Deleting an existing child record	426
Odds and ends	427
Checking for _screen.activeform	427
Dimming the Record menu pad	428
Setting tab order on a form	428
Creating OLIB (ridding MYPROC.APP)	430
Field mapping	431
Restoring the toolbar	432
Choosing between data sets	435
Developer-only controls	436
<b>SECTION IV—The Development Environment</b>	<b>439</b>
<b>Chapter 16: Customizing Your Development Environment</b>	<b>441</b>
Machine setup and installation	442
Startup configuration	443
Default directory	443

<b>Startup files</b>	<b>445</b>
The Windows Registry	445
CONFIG.FPW	446
FoxUser.DBF	446
DEFAULT.FKY	447
FoxPro.INI	447
How to use the startup files	447
Default Directory	447
Menu Builder	447
Resource File	447
Search Path	448
Startup Program	448
A word about the help file	451
Startup switches	452
<b>Developer vs. user requirements</b>	<b>452</b>
<b>Configuring development application directory structures</b>	<b>455</b>
A little bit of history, or “how we got here”	455
Developer environment requirements	457
Drive divisions	458
Root directory contents	459
Application directory contents	460
Build directory contents	462
<b>Developer utilities</b>	<b>464</b>
My own stuff	464
WW	464
Array Browser	465
DevHelp	465
OpenAll	465
Z	465
HackCX	466
NIHBSM (Not Invented Here But Still Marvelous)	467
.CHM files	467
SoftServ’s Command Window Emulator	468
Eraser	470
<b>Third-party applications</b>	<b>470</b>
CodeMine	471
DBI Technologies ActiveX controls	472
FoxAudit	473
Foxfire!	473
HTML Help Builder	474
INTL	475
Mail Manager	476
Mere Mortals Framework	476
QBF Builder	478
Stonefield Database Toolkit	478
Stonefield Query	479

<b>Stonefield Reports</b>	<b>480</b>
<b>Visual FoxExpress</b>	<b>480</b>
<b>Visual MaxFrame Professional</b>	<b>482</b>
<b>Visual Web Builder</b>	<b>482</b>
<b>West Wind Web Connection</b>	<b>483</b>
<b>xCase</b>	<b>486</b>
 <b>Chapter 17: The Component Gallery</b>	<b>487</b>
<b>A quick tour around the Component Gallery</b>	<b>487</b>
<b>Loading the Component Gallery</b>	488
<b>What are catalogs?</b>	489
<b>What's in a catalog?</b>	489
<b>Catalogs vs. folders</b>	490
<b>The contents of the default catalogs</b>	490
<b>Catalogs node</b>	490
<b>Favorites node</b>	491
<b>ActiveX Catalog node</b>	491
<b>My Base Classes node</b>	491
<b>Visual FoxPro Catalog node</b>	493
<b>Others</b>	494
<b>Where is the Component Gallery data stored?</b>	494
<b>Catalog data</b>	494
<b>Global settings</b>	496
<b>How to configure the Component Gallery so data isn't on C</b>	496
<b>Arranging components</b>	497
<b>Navigating around</b>	497
<b>Displaying existing catalogs that don't appear by default</b>	499
<b>Creating a new catalog</b>	500
<b>Adding a new folder to a catalog</b>	500
<b>Adding your own components</b>	500
<b>Making a copy of a component in your own catalog</b>	501
<b>Views: Finding things in catalogs</b>	501
<b>Using a component in your project</b>	502
<b>Summary</b>	504

<b>Chapter 18: Project Hooks</b>	<b>505</b>
<b>Accessing the project object</b>	<b>505</b>
Application object	505
Projects collection	507
Project object	507
Files collection	509
File object	509
Servers collection	509
Server object	510
What next?	511
<b>Project tools</b>	<b>511</b>
<b>Project hooks</b>	<b>514</b>
Available functions that can be intercepted	514
At the beginning of the build process	514
After the build process is finished	515
When a file is added to the Project Manager	515
When a file is modified in the Project Manager	515
When a file is removed from the Project Manager	515
When a file is executed from the Project Manager	515
When a file is dragged over or dropped on the TreeView control in the Project Manager	516
Enabling a project hook	516
Enabling a project hook for a specific project	516
Enabling a global project hook	518
Getting in and around a ProjectHook setting	518
Project hook examples	520
<b>Chapter 19: Using the Debugger</b>	<b>521</b>
<b>Configuring the Debugger</b>	<b>521</b>
Choosing a debugging frame	522
Configuring windows	523
Common settings	523
Settings for Call Stack window	524
Settings for Output window	525
Settings for Watch window	525
Settings for Locals window	526
Settings for Trace window	526
Other options	527
The five Debugger windows	527

Trace window	527
Watch window	536
Locals window	538
Call Stack window	538
Debug Output window	540
Debugger toolbar	543
Debug Frame menu	545
Event tracking	548
Using the Event Tracker	548
Going about the debugging process	550
Types of misbehavior	550
Defects in the product	550
Compile-time errors	551
Run-time errors	551
Logic errors	551
User-generated errors	553
Debugging covers developer-generated errors	553
The debugging mindset	553
A final word	556
<b>Chapter 20: Builders</b>	<b>557</b>
The Visual FoxPro Builder technology	558
Setting up your own builder in BUILDER.DBF	560
Setting up your own builder in a BUILDER property	562
Creating a user interface for a builder	563
Data-driving your builder-building process: BUILDERD	565
<b>Chapter 21: The Coverage Profiler</b>	<b>567</b>
Quick start with the Coverage Profiler	567
Basic configuration	567
Basic reports	570
Getting the Coverage Profiler to process a log file	572
Source List and Source Code panes	572
The Coverage Profiler toolbar	572
The Coverage Profiler context (shortcut) menu	572
Coverage Profiler options	574
Coverage mode vs. Profile mode	576
Zoom mode vs. Preview mode	576
Statistics	577
Find	577
Open	579
Save	579
Customizing the Coverage Profiler with Add-Ins	579

<b>Chapter 22: Using MSDN Help</b>	<b>581</b>
The MSDN Library	581
Installation	582
Opening MSDN Help	583
Features of MSDN Help	584
The Contents tab	584
The Index tab	584
The Search tab	584
The Favorites tab	587
The MSDN Library menu bar	589
Help on the Web	594
MSDN	595
The Microsoft Knowledge Base	595
 <b>SECTION V—Advanced Development Tools &amp; Techniques</b>	 <b>597</b>
<b>Chapter 23: Including ActiveX Controls in Your Application</b>	<b>599</b>
What is an ActiveX control?	599
Where—and how—do I get them?	600
In the box	600
More? You want more?	602
Installing ActiveX controls in Windows	603
Manually installing ActiveX controls in Windows	603
How do I put them into my development environment?	605
How do I use them in an application?	607
How do I ship them with an application?	609
What if it doesn't work?	610
DLL hell	611
 <b>Chapter 24: Extending VFP Through the Windows API</b>	 <b>613</b>
What is the Windows API?	613
What do functions look like in the Windows API?	614
Using the API Viewer	614
How do I access a WinAPI function?	616
Once I've called it, how do I use it?	617
Structures	618
Pointers as return values	619
Conclusion	620

<b>Chapter 25: Adding HTML Help to Your Applications</b>	<b>621</b>
<b>What is HTML Help?</b>	<b>621</b>
Index and Search tabs	623
<b>Creating an HTML Help file using HTML Help Workshop</b>	<b>625</b>
Where is HTML Help Workshop?	625
Before you create your Help project	626
Creating your HTML Help project	627
Create levels for your help file	630
Add topics underneath a section heading	632
Setting .CHM file options and building the .CHM file	633
Add index and search capability	639
<b>Including a .CHM file with your VFP application</b>	<b>644</b>
Calling a .CHM file from your VFP application	644
Creating and calling context-sensitive help topics	644
Distributing HTML Help and your .CHM file with your VFP application	645
<b>Chapter 26: Distributing Your Applications</b>	
<b>With the VFP Setup Wizard</b>	<b>647</b>
<b>What does the Setup Wizard do?</b>	<b>648</b>
<b>Are you running SP3?</b>	<b>649</b>
<b>Preparing your application</b>	<b>649</b>
<b>Running the wizard</b>	<b>651</b>
Step 1: Locate files	651
Step 2: Specify components	651
Step 3: Create disk image directory	656
Step 4: Specify setup options	656
Step 5: Specify default destination	658
Step 6: Change file settings	661
Step 7: Finish	661
Stats of the build	662
Where are the results?	662
<b>SECTION VI—Visual Studio Topics</b>	<b>665</b>
<b>Chapter 27: Weird Question Time</b>	<b>667</b>
<b>Should I or shouldn't I?</b>	<b>667</b>
<b>Where does VFP fit?</b>	<b>669</b>
<b>What is a type library?</b>	<b>670</b>
<b>What is a GUID?</b>	<b>671</b>
<b>What's the difference between early and late binding?</b>	<b>672</b>
<b>What is Visual InterDev?</b>	<b>673</b>
<b>What's a ProgID?</b>	<b>675</b>
<b>What does reentrant mean?</b>	<b>675</b>

What's the difference between "in-process servers" and "out-of-process servers"?	675
Where do I go next?	675
<b>Chapter 28: An Introduction to ADO</b>	<b>677</b>
A (very brief and very rough) history of the Microsoft data access strategy	677
OLE DB (and ADO)	678
What is ADO?	679
Where do I find ADO on my computer?	679
What are the previous versions of ADO?	680
Getting started with ADO	680
Bare-bones steps for accessing data through ADO	680
The ADO object model	692
The Connection object	694
The Recordset object	694
The Fields collection	694
The Command object	695
The Parameters collection	695
The Errors collection	695
ADO errors	695
Dealing with constants	696
So should you use ADO?	701
<b>Chapter 29: Using Microsoft Transaction Server With Visual FoxPro</b>	<b>703</b>
What is MTS and where does it come from?	703
Installing and accessing MTS	704
Creating a package	705
More information	712
<b>Chapter 30: A Visual Basic Primer for Visual FoxPro Developers</b>	<b>713</b>
The Visual Basic IDE	714
MDI versus SDI	714
What are all these windows?	714
Your first VB form	716
Adding code to a form	718
The VB Code window	718
Saving and printing projects	719
Project components and files	719
The basics of writing code	722
The Properties window, property pages, and toolbars	722
VB's version of containership	723

Slinging code	723
Procedures vs. functions	724
Option Explicit	725
Adding your own methods to forms and applications	727
A command and function summary	730
The Immediate window	730
Variables	730
Operators	731
Booleans (logicals)	733
Dates	733
Times	733
Variants	733
Constants	734
Static variables	734
Arrays	735
Functions	735
Data conversion	736
Number handling	736
Data manipulation	736
Machine and environment	737
Date and time	737
String manipulation	738
Math functions	738
Special functions	738
Commands	738
Logic structures	738
File handling	741
Program control	742
File system	742
Variables	742
Registry	743
Classes	743
N.E.C. (Not Elsewhere Classified)	743
Creating the user interface: VB forms and controls	743
Creating a form and dropping a control on it	744
VB's intrinsic controls	744
More controls	747
Incorporating data access into your VB app	748
A brief history of data access in Visual Basic	748
DAO, ADO, 123, Hike!	749
What is ADO?	749
Connections	750
Setting up a form with an ADO Data Control	750
Connecting controls to an ADO recordset	755
Revisiting the recordset idea	756
Real-world usage of ADO in VB	757

<b>Populating a combo box and a list box</b>	<b>757</b>
<b>Using form-level methods</b>	<b>757</b>
<b>Filling the combo box</b>	<b>759</b>
<b>Filling the list box</b>	<b>760</b>
<b>Populating a grid</b>	<b>762</b>
<b>Setting up the data control</b>	<b>763</b>
<b>Adding controls to your toolbox</b>	<b>763</b>
<b>Setting up the DataGrid control</b>	<b>765</b>
<b>A quick ADO command reference</b>	<b>768</b>
<b>Properties</b>	<b>769</b>
<b>Methods</b>	<b>770</b>
<b>What's this business about Visual Basic classes?</b>	<b>771</b>



# **Section I**

## **The Interactive Use of Visual FoxPro**

Visual FoxPro has several significant differences from other development tools and programming languages: its native data engine, a gorgeous object-oriented language, and a flexible and easy-to-use mechanism to switch between native and remote data sources. One other difference that is often overlooked is a throwback to Visual FoxPro's heritage as a dBASE II clone. dBASE II was the first widely used personal computer database program, and it owed its popularity to a dual personality. The one side, of course, was an easy-to-use but very powerful programming language. The other was the ability to interactively manipulate data tables as easily as spreadsheet users massaged numbers or word processors handled text. In this section I'll show you how to use Visual FoxPro's interactive abilities. Once you're used to accessing and manipulating your data within your development environment, you'll never want to switch to another language.



# Chapter 1

## Introduction to the Tool

**It has often been said, “The choice of a programming language is not a matter of life or death. It’s much more important than that.” And I have often repeated that truism, because, naturally, it’s true. However, Visual FoxPro, despite the reverential tone that will carry through most of this book, is still just an instrument to be used to build software applications. Given that point of view, I’m going to introduce you to VFP as a tool for software development. When you’re done with this chapter, you’ll understand what you can use VFP for, and how to make your way around inside of it.**

When I started using a personal computer (right around the time they invented fire, according to my kids), the computing world was a simpler place. There were “mainframes”—room-sized machines that provided processing power for hundreds or thousands of terminals throughout a company. There were “mini-computers”—refrigerator-sized machines that provided computing services to a department or a small company. And then there were our “personal computers”—typewriter- or TV-sized boxes whose processor was used by only a single person at a time.

The applications that ran on these personal computers were fairly straightforward. You had your “word processing” and “spreadsheet” applications, both of which empowered a user to automate repetitive tasks that dealt with words or numbers.

The third major genre of applications was commonly referred to as “database programs,” but there were actually two types. The first was an end-user application, much like the word processing and spreadsheet applications. The other was actually a combination of programming language and a local database storage mechanism. The market leader in the 1980s was a program written by Wayne Ratliff called dBASE.

By the end of the ‘80s, however, hardware and software lines were getting blurred. Mechanisms to connect multiple PCs in local area networks were becoming commonplace. At the same time, personal computers were getting more powerful and were able to run a wider variety of more demanding applications.

Software development techniques were evolving at the same time. In the ‘80s, the mode of application development was to write huge, monolithic applications. This approach evolved in the early ‘90s to the use of a generic framework that would then be customized and tailored for specific needs. The framework would contain elements common to each of these monolithic applications, and application-specific modules could be built on the framework for the custom solution desired.

As we approach the 21st century, the style of application development is evolving again. Developers saw how useful the reusability of a core foundation was, but the end result—executables of many megabytes in size—was inflexible. And flexibility has become more and more important as the pace of business changes has accelerated. The reusability was important, but the pieces needed to be smaller and applicable across a larger range of uses. The huge, monolithic application needed to be broken down into components.

At the same time, the use of object-oriented languages and techniques has increased to the point of becoming generally accepted and popular in many circles, thus providing the tools and conceptual framework for building components.

## **Today's computing environment**

Estimates of the size of the computer industry (including software and services) range anywhere from “a trillion dollars” to “half of the money in the universe.” Despite this pervasiveness, it’s difficult to explain to someone not in the industry what you do. This is fairly remarkable, when you think about it, because other large industries have their people easily pigeonholed. The automobile industry, for example, has mechanics, line workers, used-car salesmen, detailers, and so on. And any of those individuals can explain where they fit in the bigger picture in a manner that’s easy to explain even after three or four beers.

You’ve probably had the same experience that I have: A well-meaning relative comes up to you at a family gathering, his face takes on a serious look, and he explains, “I was working on my computer last week, and all of sudden, the screen went blank. Can you tell me why that would happen?” No hint of what type of computer, what type of software, or what he was doing. But because “you’re in computers,” he expected you to know.

The computing environment has gotten so complex that even within the industry, it can be difficult to find common ground with an industry peer. While, obviously, it would be beyond the scope (or intent) of this book to delve into the myriad areas of computing, it’s well worth examining in more detail the environment that Visual FoxPro lives in.

First of all, we’re only going to be looking at the Windows 95/98 and Windows NT platforms running on IBM-compatible personal computers. Earlier versions of FoxPro ran on other operating systems, such as Windows 3.1, DOS, UNIX, and the Macintosh OS. Second, the delivery mechanisms have expanded beyond people’s imaginations. In the early ‘80s, applications were single-user, period. When LANs became stable, multi-user applications became cutting edge and then commonplace, although there are a remarkable number of single-user systems still in use (and still being designed). As the LAN was supplemented by WANs, client-server architectures, and notebook-toting road warriors, the dispersion of data became more and more common, but the application still sat in one place.

Nowadays, however, the application is becoming distributed as well. The increased use of the Internet has changed the delivery mechanism of both the data and parts of the application. The user interface may well reside on the user’s machine while the data processing engine stays centralized on a server in a different location. This evolution has provided one final push to building applications from components.

While existing applications remain monolithic, applications being designed today look different. The data may reside on a server, and its integrity and access is protected by a data engine. The business rules of the application—the actual work the software does—is contained in a separate layer, and that layer is not only logically separate, but may physically reside somewhere other than the database server. And the data may be presented to the user through a variety of mechanisms—a rich, highly interactive front end written in a development tool, or a simple, non-interactive view of the end result through a browser.

Visual FoxPro holds a unique place in this environment. Some tools excel at serving data to a large number of requests in a secure environment. Other tools can build robust, terrifically compelling user interfaces. But only VFP plays well in all arenas. VFP has a native data engine,

which means you have to do very little work to build a LAN application with a local data store. But its extensive programming language and powerful object-oriented capabilities make it an excellent choice for building modules that do processing of any kind. Thus, VFP is capable of building complete applications that can run on a single machine, a LAN, a WAN, or a client-server infrastructure, but it's also an excellent tool for building components—particularly the middle-layer, business-rules components described earlier—that work in a distributed environment.

## Installing and starting Visual FoxPro

Visual FoxPro is one component of Microsoft's Visual Studio 6.0 package, but it's also available separately, much like Excel is available both by itself but also as part of Office. Installing VFP in either case is similar; Visual Studio enables you to install all of VS or just selected components. By default, VFP is installed in the Program Files\Microsoft Visual Studio\VFP98 directory, and places Microsoft Visual Studio 6.0 and Visual FoxPro 6.0 menu options on the Start menu.

If you haven't already, you will want to install all of the MSDN help files—and if you have the disk space (about a gigabyte), install all of MSDN, not just the VFP-specific topics.

The main program is called VFP6.EXE in the VFP98 directory. (Visual Studio 6.0 was initially going to be named Visual Studio 98, and each of the tools was going to be named with a "98" moniker as well. Evidently the "98" didn't get completely expunged, as all of the directories under Visual Studio contain a "98" as part of the tools' names.)

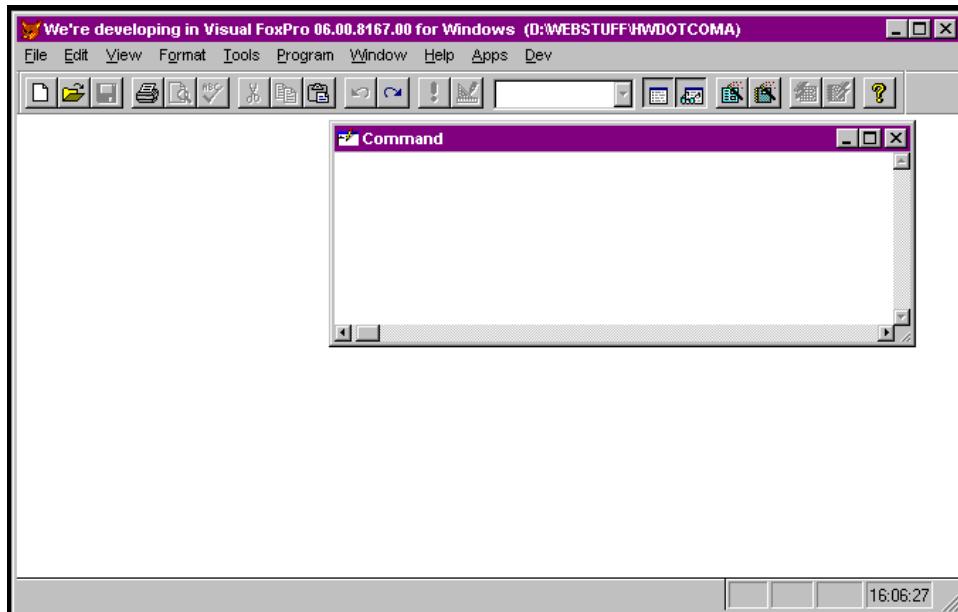
To start VFP, click Start, Microsoft Visual Studio 6.0, and then click Microsoft Visual FoxPro 6.0; or click Start and then click Microsoft Visual FoxPro 6.0. You can also put icons on the desktop and taskbar to act as shortcuts. See details in Chapter 16, "Customizing the Development Environment." Upon selecting the VFP menu option or icon, VFP will load. The first time you run Visual FoxPro, a splash screen appears and offers you a variety of choices, such as exploring sample applications or creating a new application. Each time I reinstall and then load VFP, the first thing I do is check the "Don't display this Welcome screen again" check box. (Should you ever need to, you can display this screen by running the VFP6STRT program in \Program Files\Visual Studio\VFP98 or wherever you installed VFP.) Once past the splash screen, you'll be presented with the screen in **Figure 1.1**.

## The Visual FoxPro interface

At first glance, the VFP interface might not seem very interesting: a menu, a toolbar, and an empty window named "Command." But before I go into more detail, let me mention a couple of things. I'm a big fan of customizing the environment to make my life easier, and the interface in Figure 1.1 differs from the default Visual FoxPro development environment in two ways.

First, I've customized the title bar to display the version of VFP and the current default directory. Knowing which version of VFP is running is pretty handy if you flip-flop from one version to another during the course of your day. And the current default directory is even handier—knowing whether a file is in the current directory or if you have to prepend a path before accessing it becomes second nature instead of a matter of trial and error. Second, I've added a couple of menu pads to the basic menu so I can quickly access features or functions that I use often. In effect, I'm running my own version of VFP—customized to suit the way I

work, just as you write applications customized to suit the way your users work. I'll explain how I made these changes, as well as suggest some other customizations you might want to make, in Chapter 16, "Customizing the Development Environment." I just mentioned it now because I didn't want to have to rip out all of these changes from my development environment while writing this book just to provide pristine screen shots.



**Figure 1.1.** The initial Visual FoxPro interface.

## The Command window

Remember that empty window from a minute ago? That empty window is one of the coolest features of Visual FoxPro. The Command window is used to enter commands that tell VFP what to do. You don't have to use the Command window for everything—there are menu options that correspond to many common tasks. However, you'll find that you have much more discrete control over the commands, as well as more flexibility, because the menus are limited in scope and depth. For those of you with long memories, you might remember when Windows 3.0 started to replace DOS, but some people didn't want to switch—they thought they were more efficient with a command-line interface than having to "mouse around" all day. Visual FoxPro gives us the best of both worlds—precious few other tools do that.

The VFP interface is a complete development environment (or "IDE," for Integrated Development Environment). You can create, test, debug, compile, and ship entire applications completely within this IDE—you don't need additional tools. If you've used other components of Visual Studio such as Visual C++ or Visual InterDev, you'll notice that this IDE isn't the same as that IDE for those tools. This is because of the long heritage of FoxPro—it's been around a lot longer than any of the other components of VS, and the IDE has been tuned to

provide a complete environment. However, given the move of VFP toward a component builder in Visual Studio, it wouldn't be surprising to see the VS and VFP interfaces converge.

Unlike any of the other tools in Visual Studio, the VFP IDE performs two functions. You can use Visual FoxPro to work with local databases in an interactive fashion, just as you use Excel and Word. You can open tables, add, edit and delete data, sort and query, and produce reports, as many personal computer users did on a daily basis with VFP's predecessors. Of course, it's probably overkill to use VFP just for that purpose, and, for most end users, it's probably too difficult to use strictly for interactive use. Other tools, such as Microsoft Access, are far better suited for end users to manage their personal databases.

The advantage that VFP's interactive access to data provides, however, is for the developer. Having had direct access to data during development makes life so much easier than if you had to execute a separate program in order to get at it. How many times have you wanted to run a sample query but couldn't, because you didn't know what data was in the table given you by your customer? In VFP, just open the table, look through it, and pick out what you want to use as sample data. Need to add a few records to test a new feature? Open the table and append them manually. Quick and simple. With this extra power, of course, comes responsibility. The data isn't protected from accidental changes in interactive mode—even from you! I'll discuss ways to safeguard your data from mistakes later.

Good enough? So what's the second function? Well, not only are you able to issue commands that access data from the Command window, but you are able to execute virtually any command and return the value of any expression from within the Command window. (The exceptions are primarily commands that are part of a logic structure, like IF or DO CASE.) This means you can quickly test commands and functions interactively, instead of having to (1) enter them in your program, compile, run, and see if your attempt was correct; or (2) create a dummy program for testing, and also go through a compile and run sequence just to validate the command or function. With the Command window, simply type the command, press Enter, and VFP will either execute it or respond with an error message that explains what went wrong.

Now let's move on to the rest of the interface. I'm going to assume you're fairly comfortable with the Windows interface. However, I've found that many developers don't know all of the names for the controls and functions they use every day, so I'm going to review them quickly.

## **Windows, menus, and toolbars**

A Windows 95/98 or Windows NT window has a bar at the top of the window that's a different color, contains an icon and a text string on the left, and controls at both ends. The bar is the title bar. Clicking the icon opens a menu called the Control menu. The menu options in the Control menu allow you to manipulate the window. The controls at the right of the title bar allow you to (from left to right) minimize, maximize, and close the window, effectively duplicating the Control menu options.

The string of words across the top of the screen under the title bar is called a menu. Each word is called a menu pad; clicking a menu pad or pressing the Alt key and the underlined letter in the menu pad displays a drop-down menu. Each line in a drop-down menu is called a menu bar or menu option, and can be executed by clicking it, moving the highlight to the desired menu option and pressing Enter, or pressing the underlined letter in the menu option.

If a menu pad has an exclamation mark in front of the word, selecting that menu pad won't cause a drop-down menu to display; instead, it'll launch a function directly. If a menu option has a series of three dots (an ellipsis) following the text string, selecting that menu option will bring forward a modal screen prompting the user for additional information.

Just as is the case throughout the Windows interface, pressing the right button on the mouse ("right-clicking") will open a drop-down menu at the location of the mouse pointer in many areas of Visual FoxPro. What to call this menu is a subject of heated debate at industry conferences, and will likely cause the next World War. I've heard it referred to as the "shortcut menu," the "right-click menu," and the "context menu." I'll use the last term because it describes most clearly what the menu does: provides menu options in context to the current mouse position. You can use a black marker and write in your own preference throughout this book if you like.

The string of buttons under the menu is a toolbar. Visual FoxPro has 12 different toolbars, each of which has buttons that relate to a specific area of functionality. Moving the mouse over a button in a toolbar displays a small rectangular box with a description of what the button does; this box is called a Tool Tip.

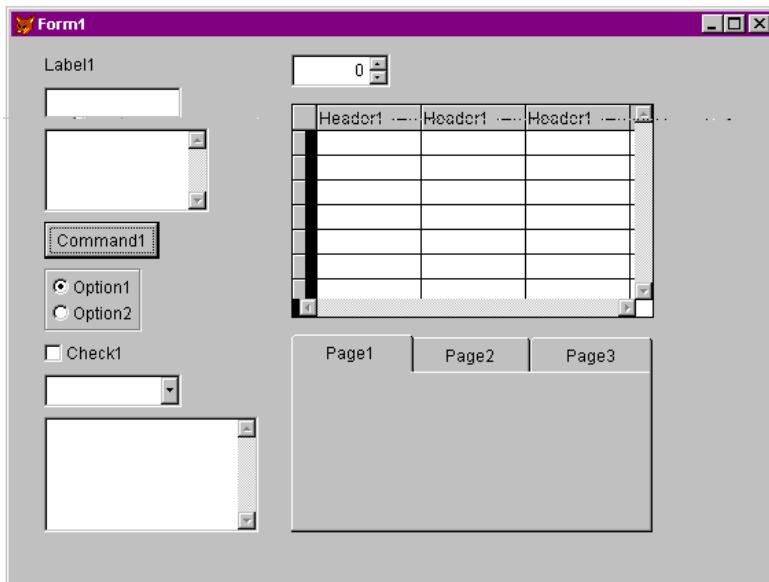
You can force a toolbar to display in several ways. First, selecting the View, Toolbars menu option will open a dialog listing all available toolbars. Check the toolbar(s) of interest and click OK, and the selected toolbars will be displayed. Right-clicking in an empty region of any open toolbar will display a toolbar context menu, with the list of all available toolbars. The toolbars already displayed will be listed with checkmarks next to them. Finally, certain toolbars are by default automatically displayed when specific functions are executed. For example, when a report is previewed, the Print Preview toolbar automatically displays.

You can customize toolbars, adding and removing buttons as well as creating your own toolbars. Details are in Chapter 16, "Customizing the Development Environment."

## Controls

Windows uses a set of objects so the user can interact with a program. Commonly used controls (but by no means all of them) are shown in **Figure 1.2**.

Navigation between controls has a number of tricks many people aren't familiar with. The currently selected control is said to "have focus" and can be identified in one of several ways, depending on which control it is. Controls that take data input will show a cursor bar in the control if it's empty, or a highlighted block of information if the control isn't empty. Controls that don't take data input will show a dashed line near the border of the control (as the control that says "Command1" in Figure 1.2.) You move from one control to the next by pressing the Tab key. This is a matter of some consternation to users of character-based programs (either on the PC or on "big iron" machines) because they expect the Enter key to move the cursor. (Note that you can often change the default behavior of a control to display the focus differently by setting a property for that control.)



**Figure 1.2.** Common controls found in the Windows interface.

Starting with the top of the left column, the following controls are displayed:

- Label: Typically read-only, a label is used to display static information to the user.
- Text Box: Allows the user to enter and edit a short string of information with a maximum number of characters. This information can be a string of text, a number, or a date.
- Edit Box: Allows the user to enter one or more lines of information, such as a paragraph of text.
- Command Button: Clicking it or pressing Enter when the button has focus allows the user to execute an action.
- Option Group: An option group is made up of one or more option buttons. This control used to be called a set of “radio buttons,” fashioned after a car radio that allowed only one button to be pushed in at a time. When the Option Group control has focus, you can move the focus within it to a specific button by using the cursor arrows. Pressing the spacebar when a button has focus selects that button (fills in the dot and removes the dot from the previously selected button). Pressing Enter will select the button that has focus and then move the focus to the next control, as if you had pressed Tab.

This is actually a bit confusing. Suppose you had a two-button option group, and the first option button had focus and was selected (the dot was filled in). Pressing Enter will just move the focus to the next option button (not the next control). Because the current button already was selected, pressing Enter didn’t change its status. Now the

focus is on the second button, but the first button is still selected. Pressing Enter now will cause the second button to be selected (the dot will get filled in), and the focus will move from the second button to the next control (the control that says Check1). I only make a big deal out of this because, someday, you will run into a user who claims your application has a bug in it because one of the option groups isn't working "properly." It is incumbent on you to understand how they're supposed to work.

- Check Box: Allows a user to select ("check") or deselect ("uncheck") it as a way of turning a flag on or off, or otherwise communicate binary information. Pressing the spacebar will change the state of the control (from checked to unchecked or vice versa) but keep the focus on the check box, while pressing the Enter key will change the state of the control and move the focus to the next control.
- Combo Box: The combo box actually has two flavors. Both of them open up (or, more precisely, "drop down") when activated to display a list of choices. One flavor—the Dropdown List—allows the user to select from the predefined list of choices. The other flavor—the Dropdown Combo—also presents an edit area where the user can enter her own data in lieu of selecting a choice in the list. Once the combo box has received focus, you can drop the list down by clicking the arrow or by pressing the spacebar. (This second mechanism is little-known, but very handy to mention when a user complains that to make a choice from the combo box she has to take her hands off the keyboard.) You can navigate to a specific choice by using the mouse or by typing the first few characters of the desired choice. (If the list of choices is sorted in alphabetical order, this works much better, but it's not an absolute necessity.) Pressing Enter when a choice in the list is highlighted selects that choice. The combo box will close and the selected choice will be displayed in the closed combo box. If you entered information into a drop-down combo, the information you entered will be displayed in the closed combo box.
- List Box: Allows the user to select from a list of choices, and differs from the drop-down combo in that more than one choice is visible at a time, and that more than one choice can be selected. Once the control has focus, the user can type the first few characters of the desired choice to highlight it, just as with the combo box. If the appropriate properties are set for the control, the user can select more than one item in the list using the mouse or, if he's a glutton for punishment, a combination of the cursor and spacebar keys, or rearrange the order of the items in the list box.
- Spinner: Allows the user to enter a numeric value or visually adjust a numeric value up or down using the arrows on the control.
- Grid: Similar to a spreadsheet, the grid allows the display (and edit) of a tabular arrangement of data. The grid is powerful because it provides a window into what could possibly be a larger table of data than would ordinarily fit on a form. The grid is almost an entity unto itself, with customizable columns, rows, and headers.
- Page Frame: Sometimes referred to as a "tabbed dialog," the page frame allows the user to easily flip between multiple pages of data on the same form without having to call up additional forms.

## Conclusion

The Visual FoxPro interface conforms to many standards of the Windows Interface Guidelines, and doesn't require a great deal of training if you're comfortable with Windows. However, the Command window is a unique feature in VFP—allowing interactive access to data as well as the execution of nearly every command and function in the language.



# Chapter 2

## The Language

All languages are judged in part by the breadth and depth of their function and keyword command set. Given these criteria, Visual FoxPro outdistances all others in both aspects, because it contains hundreds of built-in functions and a command structure that has been upgraded for 15 years. Indeed, the only complaint most people have is that the language has grown bloated—that there is simply too much to absorb and remember. In this chapter, I'm going to examine the function and command set of VFP 6.0 and point out the most important and useful components of the language.

Most people, when faced with a new programming tool, need to know two things. First, they need to understand the details of the language, and then they need to know how to assemble pieces of the language into programs. As the industry moves into component-based applications, “language” might be replaced by “components,” and “assemble pieces of the language” will be translated into “assemble components into applications,” but the basic needs are still the same.

Visual FoxPro is no different, except that there's a third item to learn: how to work with the native data engine.

This chapter will address the first of these three needs—what commands and functions you need to know about. It's simply not enough to read through the reference manual like you might have done with Fortran IV or an early version of BASICA. The Visual FoxPro language is incredibly rich, and time and time again, I've run into people who have written a 5- or 20- or 100-line routine that is intended to perform the same function as a native VFP function. The only differences, usually, are that the hand-coded routine is somewhat less reliable and measurably slower.

Before I begin, however, I'd like to make two points. I'm not going to cover the language exhaustively—that would take an entire book in and of itself—and, in fact, such a book already exists. *The Hacker's Guide to Visual FoxPro 6.0* is the exhaustive guide to every command and function in the language. My purpose is to bring programmers new to VFP up to speed on what's important, and what's not, and to update those experienced with earlier versions of FoxPro—either FoxPro 2.x or Visual FoxPro—as to what's new, and what's changed.

Second, in its most comprehensive sense, the Visual FoxPro “language” encompasses much more than just “commands and functions.” Indeed, FoxPro used to ship with a set of reference manuals, one of which was called “the Command and Function” reference—known to many wizened developers simply as “C&F.” VFP 6.0 now contains a myriad of syntax that can't be categorized simply as C&F. The VFP object model includes another thousand properties, events and methods (“PEMs” for short)—yes—in addition to the thousand commands and functions in the language! However, I'll just deal with C&F in this section, and address the PEMs later.

## Working with Visual FoxPro interactively

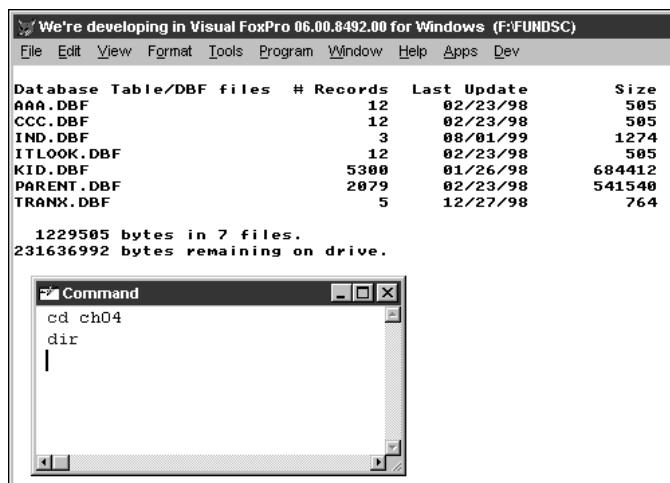
Everything I'll discuss in this chapter can be tried using the Command window—and I urge you to open VFP and play along. First, I'll explain how to use the Command window interactively.

### Using the Command window

You use the Command window to enter commands in order to perform some sort of action. In this, it's similar to a command-line interface—the DOS prompt (er, in NT, it's the “Command Prompt”) comes to mind. However, you can also use the Command window to test expressions, and it's here where we'll start.

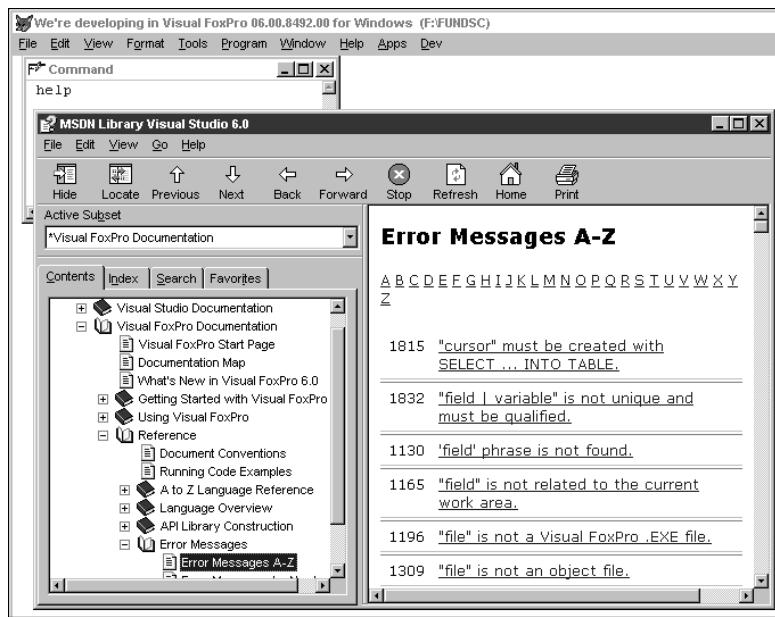
By the way, if you accidentally close the Command window, you don't have to exit VFP and start it up again—just open it using the Window, Command Window command, or by pressing Ctrl+F2. In fact, keep the keystroke combination Ctrl+F2 in mind—if the Command window is open but it's not the active window on the desktop, Ctrl+F2 will make it the active window. I do this all the time when I'm typing in a program file and want to switch to the Command window without lifting my hands from the keyboard to grab the mouse.

To execute a command or expression, type it into the Command window and press Enter. The Command window is not case sensitive, so you can use either uppercase or lowercase, or a combination of both. For example, typing “dir” in the Command window and pressing Enter will cause a list of files with the extension “DBF” to be displayed on the main FoxPro desktop. See **Figure 2.1**.



**Figure 2.1.** Entering the DIR command into the Command window.

Another example is issuing a command to run a program. For example, typing “help” and pressing Enter will bring forward the Visual Studio MSDN Library help window. See **Figure 2.2**. In this particular case, it's actually a separate application (you can see a new icon in the taskbar and the task-switching window).



**Figure 2.2.** Typing the **HELP** command in the Command window opens Visual Studio Help.

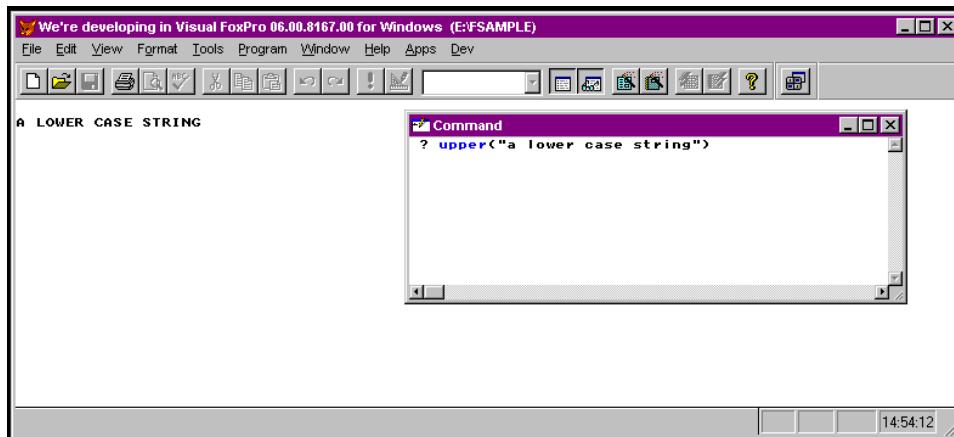
Both “DIR” and “HELP” are commands—they tell VFP to do something. However, you can work with functions and expressions in the Command window as well. The “?” command, in concert with an expression, echoes the result of that expression to the VFP desktop. For example, entering the command:

```
? upper("a lower case string")
```

into the Command window and pressing Enter will display the result of the UPPER() function being applied to the text string “a lower case string”. See **Figure 2.3**. (In between each of these screen shots, I used the CLEAR command to clear the VFP desktop of the results of the commands I’ve executed.)

### The Command window context menu

If you’ve been following along and entering these commands yourself, you’ll see that all of the commands you’ve entered will be visible in the Command window. By default, the Command window retains all of the previously entered commands, which is incredibly handy. To execute a command again, all you have to do is move your cursor to a previous command and press Enter. Furthermore, you can edit that command and execute the modified version just by pressing Enter. I remember the old days when you had to type a long command in by hand each time—quite a nuisance when the only difference between the two commands was the name of a field or a missing comma.



**Figure 2.3.** The “?” command echoes the results of expressions to the FoxPro desktop.

However, if you look at each of the preceding screen shots, you'll notice that the Command window displays only the current command. How'd I do that? One possible explanation is that I brushed up the screen shot in some image-editing program. Another explanation is that I quit VFP and started it again so that the Command window would be empty, and for years that's what presenters had to do if they wanted to have a virgin window when they started their presentations. However, in VFP 5.0 and later, a context menu was added to the Command window. If you right-click while your mouse is in the Command window, a menu will appear. You can use the Clear menu option to empty the Command window, and that's what I did in order to present cleaner screen shots for this book.

If you examine the other menu options in the Command window context menu, you'll see the standard Cut, Copy, and Paste functions, just as you would expect in any type of text or editing window. Click the Build Expression menu option to open a dialog that allows you to build VFP expressions by picking and choosing from lists of functions and operators.



*The Expression Builder dialog box opened from the Command window context menu is by default the native Visual FoxPro expression builder. You can use your own expression builder instead—the details are explained in Chapter 16, “Customizing Your Development Environment.”*

The Execute Selection menu option allows you to highlight a block of text in the Command window and execute it, all in one step. This is handy for executing several commands at once; before this functionality was available, you'd have to (1) open up an empty program file window, (2) copy the series of commands into that window, (3) save that file, and (4) run it. This was about three steps too many for most people. Furthermore, you ended up with a ton of files in the current directory with names like XXX.PRG, DELETEME.PRG, and JUNK.PRG.

The last option in the Command window context menu is Properties; selecting it opens a dialog that allows you to set various properties of the Command window. Many of the controls in the Edit Properties window are disabled for the Command window because this same dialog

is used for program file and text file windows as well, and the options don't apply to the Command window.

The drag-and-drop editing feature is one that I've overlooked until recently. When this feature is enabled, you can highlight a block of text, right-click in the middle of the highlighted block, and then drag a copy of that block to another location—either in the Command window or to another window altogether. My Ctrl+C and Ctrl+V key combinations see considerably less wear these days.



*You aren't limited to dragging within Visual FoxPro anymore. As I was writing this chapter, I didn't cut and paste (or, <gasp>, retype!) any code. Instead, I dragged code snippets from VFP into Word. This capability is called OLE Drag and Drop, and is new to VFP 6.*

## Building commands, functions, and expressions

Now that you're comfortable entering and executing commands in the Command window, it's time to look at how commands, functions, and expressions are built in VFP.

### Commands

A command is a string of text that instructs VFP to perform an action. Most commands can be executed from the Command window and from within a program, but some, such as logic structure keywords like DO CASE or ENDIF, can be used only in programs. Commands consist of a required keyword or phrase followed by one or more optional keywords and phrases. Commands can be typed in any combination of case, and many keywords can be abbreviated to the first four letters. As the language has grown, this practice has fallen out of style both because the first four letters are not unique in more and more cases, and because this capability doesn't work with the object-oriented notation used with forms and classes. The order of optional keywords generally doesn't matter (there are one or two annoying exceptions documented in Help), and so all of the following commands are identical:

```
WAIT window nowait "This is a message"
wait wind nowa "This is a message"
wait WIND nowa "This is a message"
wait wind "This is a message" NOWAIT
```

The maximum length of a command is 8,192 characters, and because most monitors available today won't display an 8,000-character string on a single line, it's common practice to split a long command into multiple lines, ending each partial line with the VFP "line continuation" character—the semicolon. This makes even more sense when you break up commands or expressions into logical segments. The first command in each of the next two examples is an illustration of a poor way to display a command; the second is a much more readable, and thus recommended, method:

```
select cNameFirst, cNameLast, cNameOrg, dLastCall, cTypeRep ;
  from COMPANY, REPS ;
  where nRegionID inlist(1,2,3,4,5) ;
  order by cTypeRep, cNameLast into cursor csrNew
```

```
select cNameFirst, cNameLast, cNameOrg, dLastCall, cTypeRep ;
  from COMPANY, REPS ;
  where nRegionID inlist(1,2,3,4,5) ;
  order by cTypeRep, cNameLast ;
  into cursor csrNew

m.lFinalAmt = iif(m.lWasSold, ;
  nJobAmt * m.nMarkup + iif(m.lWasInstalled, m.nAmtInstall, 0), ;
  m.nAmtProposed * m.nFakeChargeup)

m.lFinalAmt = ;
iif(m.lWasSold, ;
  nJobAmt * m.nMarkup + iif(m.lWasInstalled, m.nAmtInstall, 0), ;
  m.nAmtProposed * m.nFakeChargeup)
```

You can use the semicolon to continue commands across multiple lines in the Command window; while useful, it can also cause consternation if you're not paying attention. Typing a command, a semicolon, and then pressing the Enter key will not generate any results—because VFP thinks you're in the middle of a command, and it's waiting for you to finish the command with another line that doesn't end with a semicolon.

## Variables

A variable is a temporary location in memory used to store a piece of data during interactive use and program execution. Using variables in VFP is remarkably easy and extraordinarily flexible. First of all, unlike other languages, VFP takes care of all memory management. You don't have to declare memory space, make sure your variables are sized properly, or release them in order to free up memory for a future operation. VFP does all this for you. VFP also does not require you to define the data type of a variable—which is both a blessing and a curse. It's a blessing because it's considerably less work; it's a curse because it's easy to make programming errors based on the assumption that a variable is the wrong data type. For those of you used to strongly typed languages, it's kind of like getting out on the high wire without a safety net, although with somewhat less serious consequences. Fortunately, our community has adopted a number of standard techniques to reduce the risks of being weakly typed.

Variable names can be up to 254 characters long, although in practice most programmers get tired of using variable names longer than 120 or 130 characters. (Okay, I'm just kidding here—the point is that you're not limited to using cryptic names because of name-length restrictions.)

To create a variable (often called a “memory variable” in the VFP world, although I don't know why, since there wasn't ever such a thing as a “hard disk variable”), simply enter in the Command window a command like the following:

```
m.cX = "Herman"  
m.dBirth = {^1923/12/31}  
m.nAmount = 99.50  
m.iCounter = 45  
m.lLovesRockAndRoll = .t.
```

At this point, you've declared five variables, each of different types. The first is a character string, the second is a date value, the third is a numeric value, the fourth is an integer, and the fifth is a logical variable.

You'll notice that I didn't type the entire variable in the same case. Instead, for increased readability, each "word" in the variable name (except for the first character after the "m.", which I'll get to in a second) was capitalized. While "m.dBirth" and "m.dbirth" may seem to be pretty much the same, as names get long, it's easy to make a mistake. Consider:

```
m.namountinstallationtaxdefault
```

versus

```
m.nAmountInstallationTaxDefault
```

The first one kinda makes you think that Mary Poppins and Bert should include it in a song, doesn't it?

Note that the first character of each variable name is a mnemonic for the type of data it holds. Visual FoxPro doesn't enforce this—you could just as easily enter commands like:

```
m.cX = 1234.56789  
m.dBirth = "January 12"  
m.nAmount = .f.
```

and VFP wouldn't be any the wiser. The variable, m.cX, even though it starts with a "c", would be a numeric variable, m.dBirth would be a character variable, and m.nAmount would be a logical variable.

However, this convention helps you from writing code like so:

```
m.nAmount = m.nAmount + "No data entered"
```

Even though you might be new to VFP, you can already guess that this line would cause an error. The important thing to remember is that Visual FoxPro will let you enter this line and will compile it without complaining. For all VFP knows, you had already assigned another text string to the original m.nAmount variable, with the end result that the command above would just be concatenating two text strings.

The other component of a variable name that might be unfamiliar to you is the "m." prefix. Strictly speaking, this prefix is not part of the variable name. Instead, it explicitly identifies the name as a variable; if a variable has the same name as a field in an open and currently selected table, using the name without specifying the prefix will always return the value in the field. Preceding the variable name with "m." ensures that the value will be that of the variable.

Suppose you have a table, COMPANY, with a field named cNameComp, and a single record where the value of the cNameComp field is “Benefit Evaluations, Inc.” The following line will assign the string “Interactive Displays” to the memory variable m.cNameComp:

```
m.cNameComp = "Interactive Displays"
```

However, entering the following command in the Command window will not return “Interactive Displays” but rather “Benefit Evaluations, Inc.” because that’s the value in the table:

```
? cNameComp
```

Because you precede the name with an “m.”, the following command will return “Interactive Displays” instead:

```
? m.cNameComp
```

If this theory is uncomfortable, turn down the corner of this page so you can come back to it once you’re used to working with tables. If you let this idea go by without totally understanding it, your future will hold some long debugging sessions.

## Operators

Visual FoxPro has all of the standard operators you’d expect in a programming language, including addition, subtraction, multiplication, division, exponentiation, comparison ( $>$ ,  $<$ ,  $=$ ), modulus (%), and “contained in” (\$). String concatenation is performed with the plus (+) and minus (-) operators. The minus operator will trim trailing spaces from the expression preceding the minus, while the plus operator will not.

```
m.cName = "Herman" + " " + "Werke"  
m.nAmount = m.nAmount - 100  
m.nSquareFeet = m.nSide ^ 2  
m.nRemainingDonuts = m.nNumDonuts % m.nNumPeople  
m.lSteveIsIn = "Steven Tyler" $ "Steven Tyler, Joe Perry, Brad Whitford, Tom Hamilton"
```

## Functions

A function is a predefined Visual FoxPro keyword that performs an operation and returns a value. In most cases, the function will take one or more parameters, some of which may be optional. Parameters are included in the parentheses that follow a function, and multiple parameters are separated by commas. Parameters can be constants, variables or other expressions; there are limits to the number of levels that can be nested, but for most practical purposes, you shouldn’t worry about them.

 All limits can be found in the Help file. Open Visual FoxPro Documentation, Using Visual FoxPro, Programmer’s Guide, Appendix, and then read the “Visual FoxPro System Capacities” help topic.

---

Functions can be categorized into more than a dozen broad areas, including string manipulation, date and time handling, mathematical operations, database, table and field mechanisms, file management, low-level file handling, and array and memory variable manipulation.

Specific functions are covered in detail later in this chapter.

## Expressions

An expression is the result of the combination of one or more constants, variables, and functions, all combined with operators. The general syntax of an expression is of the form:

```
expression operator expression
```

where each side of the operator must be of the same type. In other words, you can add a string to a string or compare two dates to determine which is more recent, but you can't put a logical on one side and a number on the other.

The following lines each represent a single expression:

```
"I am a string"  
123.45  
100 + 200  
m.nAge * 12  
m.nAmount > 999.99  
str(m.nAmount,10,2)  
"Your account balance is " + str(m.nAmount,10,2)
```

The first two expressions are character and numeric constants, respectively. The third is an arithmetic expression that evaluates to the numeric value 300. The fourth is another arithmetic expression that multiplies a memory variable and a numeric constant. The fifth appears to be a comparison of two expressions, but the whole string evaluates to a logical value—which might be either true or false, depending on the value of m.nAmount.

The second to the last expression uses a function to convert a numeric variable to a text string (the 10 and the 2 control how long the text string will be and how many decimal places will be in the return value). The last expression concatenates a character constant with the function expression from the previous line.

You can also use the AND, OR, and NOT keywords to combine and negate multiple expressions. These expressions are called compound expressions. For example, the OR operator will return a logical true if one side of an expression or the other is true:

```
? (m.nAge > 40) or (m.cHairColor = "Grey")
```

You could store the value of this compound expression to a memory variable just as easily:

```
m.lIsAFossil = (m.nAge > 40) or (m.cHairColor = "Grey")
```

The AND operator will return a logical true only if both sides of an expression are true:

```
m.lIsASexSymbol = (m.cHairColor = "Bald") and (m.lHasPocketProtector = .t.)
```

I'd like to point out that the parentheses around each of the two expressions aren't required—I've included them to enhance readability. Second, you don't have to equate a logical expression with a .t. or .f., because the logical expression itself is—yes—a logical expression and can be evaluated. Thus, the following expression is just as valid:

```
m.lIsASexSymbol = m.cHairColor = "Bald" and m.lHasPocketProtector
```

Finally, Visual FoxPro short circuits the evaluation of AND and OR expressions so that only the first condition may be evaluated if that's all that's necessary. For example, if the first part of an AND expression is false, the rest is ignored, since whether the rest is true or false doesn't matter. Similarly, if the first part of an OR expression is true, the rest is ignored because, again, it doesn't matter.

All the parts of an expression must be of the same type. When concatenating multiple expressions with AND or OR, each of the parts must evaluate to a logical value. It's common to run into a "data type mismatch" or an "operator/operand type mismatch" error message—this is the case when part of an expression doesn't match the rest. To resolve it, break down each part separately and make sure it's returning the correct type of data.

## A Visual FoxPro command and function overview

974. That's how many commands and functions I've found. (It's not simply as easy as counting through Help's A to Z Language Reference. They've missed a few and omitted a bunch more.) I've been using this language for nearly 20 years, but in putting this chapter together, I still found a few that I've never seen before. How are you—whether you're new to Fox completely, or at least new to VFP—going to absorb and get your arms around nearly 1,000 commands and functions? (And then do it again in a few chapters when I hit you with a thousand PEMs?)

Well, as a wise hunter once remarked, "How do you eat an elephant? First, you cut it up into big pieces. Then you cut those big pieces into very small pieces." Well, maybe it was a wise chef, but you get the idea.

Similarly, I flagged a number of these thousand commands and functions as obsolete. A keyword might be obsolete because it's simply not needed anymore, or because there's a better way to do the same thing, or because there's a new way (which, actually, is a little different than "better"). It might also be obsolete because Microsoft said so. In any case, if you see a language element in the Help file (or in someone's code) that isn't listed here, you should probably stay away from it.

Once I winnowed this enormous list down some, I divided the survivors into about 30 different categories. Obviously, any attempt at categorization is somewhat arbitrary. Here's how I decided on these groupings.

---

I created the categories using a number of logical (and, hopefully, obvious) groups. In many cases, I ended up with a manageable number of entries in the group—I figured 20 was about the most someone could absorb in a single category.

With others, though, there were still 50 or 60—so I subdivided the groups further. For example, under the grouping of “data,” I subdivided the keywords according to databases, tables, indexes, and so on.

I didn’t put any keyword into more than one group. Yeah, you could make a pretty good case that a function like DTOS (date to string conversion) should belong in both the “date/time” and “string” categories. I used the criteria “How would I use this thing?” In other words, “What operation am I doing that I need to look in my toolbox and find a function to do that?”

In each category, I’ve highlighted the big ones, and have provided a quick and dirty description of when you might find these useful. Remember, these descriptions are not exhaustive—see the Online Help (or better, the *Hacker’s Guide*) for the gory details. You’ll want to consult the *Hacker’s Guide*, particularly, for all the tips and tricks—and there are loads of them!

I’ve also indicated when each command or function came about—starting with FoxPro 2.6—so that those of you who are old Fox folk can quickly review “what’s new.” For example, if you see a “Y” in the column “In 3”, the command was introduced in Visual FoxPro 3.0, even though there isn’t a “Y” in “In 5” and “In 6.” The abbreviation “Ob” stands for “obsolete.”



Finally—and this is the cool part—with the book’s source code, I’ve included a .DBF that contains this listing. You can sort and filter on command name, when the command was brought into the language, and what category I attached it to. That means that you can query and modify this list to suit your own needs as desired. For example, you could sort on commands strictly alpha, or just view those in a single group, or see just those that were introduced in a particular version.

Note that I’ve used parentheses to indicate that a keyword is a function. Thus, FOR(), SEEK(), and TYPE() are all functions, while FOR, SEEK, and TYPE are all commands. (And, yes, these are all legitimate examples.)

## Arrays

An array is a one- or two-dimensional structure of variables. You’ll find yourself working with arrays a good deal in Visual FoxPro—and here are the commands that manipulate arrays. This list does not include commands that stuff their results into an array, nor those commands that read from an array as a source object.

DECLARE and DIMENSION are identical. Many developers use DECLARE to initially define an array, and DIMENSION to change the dimensions of an existing array. This way, they have a clue that an array already existed when they run into a DIMENSION command. Other developers don’t use DECLARE because it is also used in the DECLARE DLL command.

I use ALEN and ASCAN a lot. The first, used with a “1” as a second parameter, tells me how many rows an array has, like so:

```
m.lnNumThings = alen(aThings,1)
```

Note that I used an “a” as the leading character of the array name to remind me it’s an array. ASCAN finds an element in an array, regardless of which row or column it’s in. You’ll want to remember that the setting of EXACT affects how the search works.

```
m.lnElementNumber = ascn(aMyArray, m.lcSomeValue)
```

ASCAN returns the element number in the array. If m.lcSomeValue is in the second column of the second row of a 5x5 array, the element number returned would be 7.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	ACOPY()	Copies an array.
N	Y	N	N	N	ADEL()	Deletes a row or column from an array.
N	Y	N	N	N	AELEMENT()	Returns the element number of an array.
N	Y	N	N	N	AINS()	Inserts a row or column into an array.
N	Y	N	N	N	ALEN()	Returns length (rows or columns) of an array.
N	Y	N	N	N	ASCAN()	Looks for information in an array.
N	Y	N	N	N	ASORT()	Sorts an array.
N	Y	N	N	N	ASUBSCRIPT()	Returns subscript values of an element.
N	Y	N	N	N	DECLARE	Defines an array.
N	Y	N	N	N	DIMENSION	Defines an array.
N	Y	N	N	N	EXTERNAL	Warns the project manager about an undefined reference.

## Client/Server

This list of commands and functions covers all of the functionality involved in talking to the back-end data source of a client/server application, and that you’d need when using transactions. Strictly speaking, transactions don’t have to be used with a client/server application, but that’s where I figured most people would look.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	N	Y	N	N	_TRIGGERLEVEL	Contains a read-only numeric value indicating the current trigger procedure nesting level.
N	N	Y	N	N	BEGIN TRANSACTION	Begins a transaction.
N	N	Y	N	N	CREATE CONNECTION	Creates a connection in the current database.
N	N	Y	N	N	DELETE CONNECTION	Deletes a connection from the current database.
N	N	Y	N	N	DISPLAY CONNECTIONS	Displays connections in a database.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	N	Y	N	N	RENAME CONNECTION	Renames a connection in a database.
N	N	Y	N	N	ROLLBACK	Sets the values of changes made during a transaction back to their original values.
N	N	Y	N	N	SQLCONNECT	Establishes a connection to a data source.
N	N	N	Y	N	SYS(3053)	ODBC Environment Handle.
N	N	Y	N	N	TXNLEVEL()	Returns a numeric value indicating the current transaction level.
N	N	Y	N	N	SQLCANCEL()	Requests cancellation of an executing SQL statement.
N	N	Y	N	N	SQLCOMMIT()	Commits a transaction.
N	N	Y	N	N	SQLCOLUMNS()	Creates a cursor containing information about columns.
N	N	Y	N	N	SQLDISCONNECT()	Terminates a connection to a data source.
N	N	Y	N	N	SQLEXEC()	Sends a SQL statement to a data source to be processed.
N	N	Y	N	N	SQLGETPROP()	Returns current or default settings for a connection.
N	N	Y	N	N	SQLMORERESULTS()	Copies another result set to a cursor if available.
N	N	Y	N	N	SQLPREPARE()	Prepares a SQL statement to be executed by SQLEXEC.
N	N	Y	N	N	SQLROLLBACK()	Cancels changes made during a transaction.
N	N	Y	N	N	SQLSETPROP()	Sets properties for a connection.
N	N	Y	N	N	SQLSTRINGCONNECT()	Establishes a connection to a data source.
N	N	Y	N	N	SQLTABLES()	Creates a cursor containing the names of the tables in a data source.

## Colors

I generally don't use these because I let the user deal with colors through Windows, as they should be doing. Your mileage may vary, as your needs may differ.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	N	Y	N	N	RGB()	Returns a single color value from a set of color components.
N	Y	N	N	N	RGBSCHEME()	Returns an RGB color pair or list from a color scheme.
N	Y	N	N	N	SCHEME()	Returns a color pair or list from a color scheme.
N	Y	N	N	N	SET PALETTE	Specifies whether the Visual FoxPro default color palette is used.

## Data – Database

These commands and functions deal with the Visual FoxPro database container and its components. Specific functionality involved with those components—for example, tables or indexes—are grouped separately under other Data subgroups.

SET DATABASE is pretty important—it's like selecting a work area that holds a table. If you have more than one database open, you often need to specify which database you want an operation to apply to. You can do that with SET DATABASE. DBC() will tell you if you have to use SET DATABASE by indicating which database is current. VALIDATE DATABASE is one of those commands you hope you never have to use, but it's handy to have.

Many of these commands are more useful if you're building database maintenance routines—utilities that you'll use to support an application, rather than utilities used to actually run an application. I wouldn't know myself; I've used the Stonefield Database Toolkit—a third-party data dictionary—that does much of that type of work for me.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	N	Y	N	N	ADATABASES()	Stuffs all open database names (and paths) into an array.
N	N	Y	N	N	ADBOBJECTS()	Stuffs all database objects (tables, views, etc.) into an array.
N	N	Y	N	N	APPEND PROCEDURES	Adds stored procedures from a text file to a database's stored procedures.
N	N	Y	N	N	CLOSE DATABASE	Closes the current database.
N	N	Y	N	N	COPY PROCEDURES	Copies contents of a stored procedure to a text file.
N	N	Y	N	N	CREATE DATABASE	Creates a database.
N	N	Y	N	N	CREATE TRIGGER	Creates a trigger for a table.
N	N	Y	N	N	DBC()	Returns the name and path of the current database.
N	N	Y	N	N	DBGETPROP()	Returns a property from the current database.
N	N	Y	N	N	DBSETPROP()	Sets a property for the current database.
N	N	Y	N	N	DBUSED()	Determines if a database is open.
N	N	Y	N	N	DELETE DATABASE	Deletes a database.
N	N	Y	N	N	DELETE TRIGGER	Deletes a trigger.
N	N	Y	N	N	DISPLAY DATABASE	Displays information about the current database.
N	N	Y	N	N	INDBC()	Determines if a database object is in a DBC.
N	N	Y	N	N	OPEN DATABASE	Opens a database.
N	N	Y	N	N	PACK DATABASE	Permanently removes all records marked for deletion.
N	N	Y	N	N	SET DATABASE	Specifies the current database.
N	N	Y	N	N	SET DATASESSION	Activates the specified form's data session.
N	N	Y	N	N	VALIDATE DATABASE	Ensures that the locations of tables and indexes in the current database are correct.

## Data – Editing

Just about every database application I've ever developed has been used to make changes to the data within it. That's not always a requirement, of course—in the case of Executive Information Systems or Data Warehouses, you could very well just be querying existing sets. But even then, someone else is likely writing pieces to somehow update the data set.

Some of these are for interactive use—EDIT and CHANGE display a table in spreadsheet or file-card fashion. Others are useful both during interactive use and for controlling how the application appears to the user. SET CONFIRM can be used to require the user to press Enter to exit a text box even after they've completely filled the text box with data.

Others are used strictly for programming. In the olden days (about four years ago), FoxPro applications often used a methodology where data was copied from tables to sets of memory variables, and those memory variables were mapped to the user interface. SCATTER and GATHER were used for moving data to and from tables, and are not as widely used as in the past. Nowadays, they're still handy for working with cursors, but most Visual FoxPro applications bind tables directly to data entry forms, and thus don't need this intermediate layer. Instead, VFP uses an internal buffer that works the same way as the programmer-created variables did in the past. OLDVAL and CURVAL are used to determine whether a record has been changed, and what the original and current values are.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	APPEND	Adds a record to a table.
N	Y	N	N	N	APPEND FROM	Adds records to a table from another file.
N	Y	N	N	N	APPEND FROM ARRAY	Adds records to a table from an array.
N	Y	N	N	N	BLANK	Clears data from one or more fields (different than stuffing .f. or zeros).
N	Y	N	N	N	BROWSE	Browse window.
N	Y	N	N	N	CHANGE	Displays a record for editing.
N	Y	N	N	N	CLOSE MEMO	Closes the window associated with a memo field.
N	N	Y	N	N	CURVAL()	Returns field values from disk for a data source.
N	Y	N	N	N	DELETE	Sets the delete flag for records.
N	Y	N	N	N	EDIT	Displays fields for editing.
N	Y	N	N	N	FLUSH	Saves table and index data to disk.
N	Y	N	N	N	GATHER	Updates table from matching variables.
N	Y	N	N	N	MODIFY	Similar to CREATE.
N	N	Y	N	N	OLDVAL()	Returns original field values for fields that have been modified but not updated.
N	Y	N	N	N	RECALL	Unmarks records flagged for deletion.
N	Y	N	N	N	REPLACE	Updates fields in a table.
N	Y	N	N	N	SCATTER	Creates memory variables from fields.
N	Y	N	N	N	SET AUTOSAVE	Determines whether Visual FoxPro flushes data buffers to disk when you exit a READ or return to the Command window.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	SET CARRY	Determines whether Visual FoxPro carries data forward from the current record to a new record created with INSERT, APPEND, and BROWSE.
N	Y	N	N	N	SET CONFIRM	Specifies whether the user can exit a text box by typing past the last character in the text box.
N	Y	N	N	N	SET SAFETY	Determines whether Visual FoxPro displays a dialog box before overwriting an existing file. Note that despite what Online Help says, the Table Designer will display a dialog asking if you want rules checked regardless of the setting of this command, and ALTER TABLE will always check rules, regardless of the setting of this command unless you include the NOVALIDATE clause.

## Data – Fields

These commands and functions are used to work with fields in a table.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	AFIELDS()	Stuffs table structure information into an array; returns number of fields.
N	Y	N	N	N	CLEAR FIELDS	Clears field list (from SET FIELDS).
N	Y	N	N	N	FCOUNT()	Returns the number of fields in a table.
N	Y	N	N	N	FIELD()	Returns a field name when passed a number.
N	Y	N	N	N	SET FIELDS	Specifies which fields in a table can be accessed.

## Data – Indexes

Indexes, perhaps more than any other feature, help distinguish Visual FoxPro from other tools. These commands and functions enable you to create, change, and otherwise query and manipulate indexes in a thousand ways.

The one command you want to stay away from until you completely understand it, and have a very good reason to work with it, is SET UNIQUE. This command forces indexes that are created to contain only a single placeholder for each unique value in the index expression. The trouble comes when the record mapped to that placeholder is deleted. The placeholder is not automatically moved to another record that shares that index value; it's simply deleted. SET UNIQUE can be handy for creating temporary indexes on the fly for special purposes, but is not suitable for day-to-day use.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	N	Y	N	N	CANDIDATE()	Determines if a tag is a candidate.
N	Y	N	N	N	CDX()	Returns the name of a .CDX file.
N	Y	N	N	N	CLOSE INDEXES	Closes all non-structural indexes.
N	Y	N	N	N	COPY INDEXES	Creates compound index tags from corresponding .IDX files.
N	Y	N	N	N	COPY TAG	Creates an .IDX file from a compound index tag.
N	Y	N	N	N	DELETE TAG	Deletes a tag from a compound index file.
N	Y	N	N	N	DESCENDING()	Returns true if the current index tag is descending.
N	Y	N	N	N	FOR()	Returns an index filter expression.
N	Y	N	N	N	IDXCOLLATE()	Returns the collation sequence for an index or tag.
N	Y	N	N	N	INDEX	Creates an index file.
N	N	N	N	Y	INDEXSEEK()	Performs a SEEK without moving the record pointer.
N	Y	N	N	N	KEY()	Returns the index key for an index expression.
N	Y	N	N	N	NDX()	Returns the name of the open index file.
N	Y	N	N	N	ORDER()	Returns the name of the index file or tag.
N	N	Y	N	N	PRIMARY()	Determines if an index tag is defined as primary.
N	Y	N	N	N	REINDEX	Rebuilds open index files.
N	Y	N	N	N	SET COLLATE	Specifies a collation sequence for character fields in subsequent indexing and sorting operations.
N	Y	N	N	N	SET INDEX	Opens one or more index files for use with the current table.
N	Y	N	N	N	SET KEY	Specifies access to a range of records based on their index keys.
N	Y	N	N	N	SET ORDER	Designates a controlling index file or tag for a table.
N	Y	N	N	N	SET UNIQUE	Specifies whether records with duplicate index key values are maintained in an index file.
N	Y	N	N	N	TAG()	Returns a tag name from an open, multiple-entry compound .CDX index file or the name of an open, single-entry .IDX index file.
N	Y	N	N	N	TAGCOUNT()	Returns the number of .CDX compound index file tags and open .IDX single-entry index files.
N	Y	N	N	N	TAGNO()	Returns the index position for .CDX compound index file tags and open .IDX single-entry index files.
N	Y	N	N	N	UNIQUE()	Returns .T. if the specified index was created with the UNIQUE keyword of INDEX or with SET UNIQUE ON.

## Data – Locking

This list contains mechanisms that allow you to attempt to lock tables or records, or to determine if locks have been placed. This distinction is important—for example, the RLOCK() function does not automatically lock a record; it just attempts to do so, and returns a value that tells whether the lock worked. However, you might not want to lock a record just to see if it's locked—so you can use ISRLOCKED instead.

SET MULTILOCKS is an important command—it allows locks to be placed on more than one record at the same time and is required if you use buffering and/or views. Otherwise, locking record 2 will release the lock on record 1.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	FLOCK()	Tries to lock a table.
N	N	N	Y	N	ISFLOCKED()	Determines whether a table is locked.
N	N	N	Y	N	ISRLOCKED()	Determines whether a record is locked.
N	Y	N	N	N	RLOCK()	Attempts to lock records.
N	Y	N	N	N	SET LOCK	Enables or disables automatic file locking in certain commands.
N	Y	N	N	N	SET MULTILOCKS	Determines whether you can lock multiple records using LOCK( ) or RLOCK( ).
N	Y	N	N	N	SET REFRESH	Determines whether and how often a Browse window is updated with changes made to records by other users on the network.
N	Y	N	N	N	SET REPROCESS	Specifies how many times and for how long Visual FoxPro attempts to lock a file or record after an unsuccessful locking attempt.
N	N	N	Y	N	SYS(3051)	Set Lock Retry Interval.
N	N	N	Y	N	SYS(3052)	Override SET REPROCESS Locking.
N	Y	N	N	N	UNLOCK	Releases a record lock, multiple record locks, or a file lock from a table, or releases all record and file locks from all open tables.

## Data – Memo Fields

These commands allow you to add data to a memo field. Commands that export data from memo fields (generally, to files) are listed under the File – Write category.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	APPEND GENERAL	Stuffs an OLE object into a table's general field.
N	Y	N	N	N	APPEND MEMO	Stuffs a text file into a table's memo field.

## Data – Navigation

FoxPro uses the concept of a “record pointer”—a flag that points to a single record, much like the cursor in a spreadsheet points to a single cell—during manipulation of a table. These commands and functions allow you to move the record pointer through a table in a variety of ways.

SEEK and LOCATE look for a record that matches certain criteria and move the record pointer accordingly. GO, SKIP, and CONTINUE physically move the record pointer through the file, and the BOF() and EOF() functions tell you whether the pointer is at the beginning or end of the file.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	BOF()	Beginning of file flag.
N	Y	N	N	N	CONTINUE	Continues the previous Locate.
N	Y	N	N	N	EOF()	End of file flag.
N	Y	N	N	N	FOUND()	Returns true if a record search worked.
N	Y	N	N	N	GO	Moves the record point in a table.
N	Y	N	N	N	LOCATE	Looks for the first match of an expression in a table.
N	Y	N	N	N	LOOKUP()	Searches a table for the first record with a field matching an expression.
N	Y	N	N	N	SEEK	Moves the record pointer to the first occurrence of a record whose index key matches an expression.
N	Y	N	N	N	SEEK()	Returns a logical value depending on whether a SEEK was successful.
N	Y	N	N	N	SET NEAR	Determines where the record pointer is positioned after FIND or SEEK unsuccessfully searches for a record.
N	Y	N	N	N	SKIP	Moves the record pointer in a table.

## Data – SQL commands

With FoxPro 2.0, elements of the SQL language were included. These language elements have been expanded and improved with each subsequent version.

I don’t know of an experienced developer who doesn’t shake their head when they run into another developer who has stayed away from learning SQL. It’s fast, simple to learn, and wonderfully powerful. Even better, the skills you learn are largely transportable—VFP’s SQL implementation is standard SQL.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	_TALLY	Contains the number of records processed by the most recently executed table command.
N	N	Y	N	N	ALTER TABLE	SQL command to modify the structure of a table.
N	Y	N	N	N	CREATE CURSOR (SQL)	Creates a cursor.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	CREATE TABLE (SQL)	Creates a table via SQL.
N	N	Y	N	N	DELETE (SQL)	Sets the delete flag for records (SQL).
N	Y	N	N	N	INSERT (SQL)	Standard SQL Insert. Adds a record to a table.
N	Y	N	N	N	SELECT (SQL)	Standard SQL SELECT command to retrieve data.
N	Y	N	N	N	SET ANSI	Determines how comparisons between strings of different lengths are made with the = operator in Visual FoxPro SQL commands.
N	N	Y	N	N	SET NULL	Determines how null values are supported by the ALTER TABLE, CREATE TABLE, and INSERT - SQL commands.
N	Y	N	N	N	SET OPTIMIZE	Enables or disables Rushmore optimization.
N	N	N	Y	N	SYS(3054)	Rushmore Optimization Level.
N	N	N	N	Y	SYS(3055)	FOR/WHERE Clause Complexity.
N	N	Y	N	N	UPDATE (SQL)	Updates records in a table with new values.

## Data – Structure

These commands allow you to create, modify, and use the structure of a table.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	COPY STRUCTURE	Makes a duplicate of a table with the same structure but no records.
N	Y	N	N	N	COPY STRUCTURE EXTENDED	Creates a table that contains the structure of a table.
N	Y	N	N	N	CREATE	Creates a table.
N	Y	N	N	N	CREATE FROM	Creates a table using a COPY STRUCTURE EXTENDED definition.
N	Y	N	N	N	DISPLAY STRUCTURES	Displays the structure of a table.

## Data – Tables

These commands provide a wide variety of tools for working with tables.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	N	Y	N	N	ADD TABLE	Adds a free table to a database.
N	Y	N	N	N	ALIAS()	Returns the table alias of a work area.
N	N	Y	N	N	AUSED()	Stuffs table aliases and work areas into an array.
N	N	Y	N	N	CLOSE TABLES	Closes tables.
N	Y	N	N	N	CREATE QUERY	Creates a query.
N	N	Y	N	N	CREATE SQL VIEW	Creates a view.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	CREATE VIEW	Creates an environment view file. Many consider this command obsolete, but if you ever want to create an environment view, there's no other way to do it.
N	N	N	Y	N	CREATEOFFLINE()	Creates an offline view.
N	N	Y	N	N	CURSORGETPROP()	Returns property settings for a table.
N	N	Y	N	N	CURSORSETPROP()	Sets property settings for a table.
N	Y	N	N	N	DBF()	Returns the name of a table open in a work area.
N	N	Y	N	N	DELETE VIEW	Deletes a SQL view.
N	Y	N	N	N	DELETED()	Returns true if the current record has the delete flag set.
N	N	Y	N	N	DISPLAY TABLES	Displays information about the tables in a database.
N	N	Y	N	N	DISPLAY VIEWS	Displays information about SQL views in a database.
N	N	Y	N	N	DROP TABLE()	Deletes a table from a database and disk.
N	N	Y	N	N	DROP VIEW()	Deletes a SQL view from a database.
N	N	N	Y	N	DROPOFFLINE()	Discards changes to an offline view and puts it back online.
N	Y	N	N	N	FILTER()	Returns the filter applied to a table.
N	N	Y	N	N	FREE TABLE	Removes database backlink from a table header.
N	N	Y	N	N	GETFLDSTATE()	Returns a numeric value identifying which fields have been changed.
N	N	Y	N	N	GETNEXTMODIFIED()	Returns a record number for the next modified record in a buffered entity.
N	Y	N	N	N	HEADER()	Returns the number of bytes in the header of a table.
N	N	Y	N	N	ISEXCLUSIVE()	Determines whether a database is opened exclusively.
N	Y	N	N	N	ISREADONLY()	Determines whether a table is opened read-only.
N	Y	N	N	N	PACK	Permanently removes all records marked for deletion and compacts the associated memo file.
N	Y	N	N	N	RECCOUNT()	Returns the number of records in the current table.
N	Y	N	N	N	RECNO()	Returns the number of the current record.
N	Y	N	N	N	RECSIZE()	Returns the width of a record.
N	N	Y	N	N	REFRESH()	Refreshes data in an updatable SQL view.
N	Y	N	N	N	RELATION()	Returns a specified relation between two tables.
N	N	Y	N	N	REMOVE TABLE	Removes a table from a database.
N	N	Y	N	N	RENAME TABLE	Renames a table in a database.
N	N	Y	N	N	RENAME VIEW	Renames a SQL view in a database.
N	N	Y	N	N	REQUERY()	Reruns a query for a SQL view.
N	Y	N	N	N	SELECT	Changes to a specified work area.
N	Y	N	N	N	SELECT()	Returns the number of a work area.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	SET BLOCKSIZE	Specifies how Visual FoxPro allocates disk space for the storage of memo fields.
N	Y	N	N	N	SET DELETED	Specifies whether Visual FoxPro processes records marked for deletion and whether they are available for use in other commands.
N	Y	N	N	N	SET EXLCUSIVE	Specifies whether Visual FoxPro opens table files for exclusive or shared use on a network.
N	Y	N	N	N	SET FILTER	Specifies a condition that records in the current table must meet to be accessible.
N	Y	N	N	N	SET RELATION	Establishes a relationship between two open tables.
N	Y	N	N	N	SET RELATION OFF	Breaks an established relationship between the parent table in the currently selected work area, and a related child table.
N	Y	N	N	N	SET SKIP	Creates a one-to-many relationship among tables.
N	Y	N	N	N	SET VIEW	Opens or closes the Data Session window or restores the Visual FoxPro environment from a view file.
N	N	Y	N	N	SETFLDSTATE()	Assigns a field or deletion state value to an object in a table.
N	Y	N	N	N	SYS(2012)	Memo Field Blocksize.
N	N	Y	N	N	SYS(2029)	Table Type.
N	N	Y	N	N	TABLEREVERT()	Discards changes made to a buffered row or a buffered table or cursor and restores the OLDVAL( ) data for remote cursors and the current disk values for local tables and cursors.
N	N	Y	N	Y	TABLEUPDATE()	Commits changes made to a buffered row or a buffered table or cursor.
N	Y	N	N	N	TARGET()	Returns the alias of a table that is the target for a relation as specified in the INTO clause of SET RELATION.
N	Y	N	N	N	USE	Opens a table and its associated index files, or a SQL view.
N	Y	N	N	N	USED()	Determines if an alias is in use or a table is open in a specific work area.
N	Y	N	N	N	ZAP	Removes all records from a table, leaving just the table structure.

## **Date****Time**

These commands and functions work with dates and times in one fashion or another.

Some determine the current date, time, day of week, and so on. For example, DATE returns the current date, while DOW returns the numeric day of the week from a date expression. Others determine the distance from one date to another, such as GOMONTH.

Because date expressions are a distinct data type, they can't be combined with character expressions. Some of these functions convert dates and times to strings and vice versa. The character string format that is used sometimes depends on environmental settings. For example, CTOD( ) can create ambiguous Date values depending on the settings of SET DATE and SET CENTURY at the time of program compilation.

There are other commands that allow you to handle Year 2000 issues in the manner you want. Finally, functions to work with Julian dates are also included in this list.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	CDOW()	Character Day of Week.
N	Y	N	N	N	CMONTH()	Returns name of month.
N	Y	N	N	N	CTOD()	Converts a character string to a date expression.
N	N	Y	N	N	CTOT()	Converts a character string to a time expression.
N	Y	N	N	Y	DATE()	Returns system date.
N	N	Y	N	Y	DATETIME()	Returns system date and time.
N	Y	N	N	N	DAY()	Returns numeric day of month.
N	Y	N	N	N	DMY()	Returns character string like "1 January 2000" from a date or datetime expression.
N	Y	N	N	N	DOW()	Returns numeric day of the week from a date or datetime expression.
N	Y	N	N	N	DTOC()	Converts a date or datetime expression to a character expression like "1/1/2000".
N	Y	N	N	N	DTOS()	Converts a date or datetime expression to a character expression like "20000101".
N	N	Y	N	N	DTOT()	Converts a date expression to a datetime expression.
N	Y	N	N	N	GOMONTH()	Returns a date that is N months from a specified expression.
N	N	Y	N	N	HOUR()	Returns the hour from a date/time expression.
N	Y	N	N	N	MDY()	Returns a date formatted like "January 1, 1980".
N	N	Y	N	N	MINUTE()	Returns the minutes from a date/time expression.
N	Y	N	N	N	MONTH()	Returns the month number from a date/time expression.
N	N	Y	N	N	SEC()	Returns the seconds of a date/time expression.
N	Y	N	N	N	SECONDS()	Returns the number of seconds that have elapsed since midnight.

Ob	In 2.x	In 3	In 5	In 6	Command/Function	Description
N	Y	N	N	N	SET CENTURY	Determines whether VFP displays the century portion of date expressions and how it interprets dates that specify only two-digit years.
N	Y	N	N	N	SET CLOCK	Determines whether Visual FoxPro displays the system clock and specifies the clock location on the main Visual FoxPro window.
N	Y	N	N	N	SET DATE	Specifies the format for the display of date and datetime expressions.
N	N	Y	N	N	SET FDOW	Specifies the first day of the week.
N	N	Y	N	N	SET FWEEK	Specifies the requirements for the first week of the year.
N	Y	N	N	N	SET HOURS	Sets the system clock to a 12- or 24-hour time format.
N	N	Y	N	N	SET SECONDS	Specifies whether seconds are displayed in the time portion of a datetime value.
N	N	N	N	Y	SET STRICTDATE	Specifies if ambiguous date and datetime constants generate errors.
N	Y	N	N	N	SYS( 1)	Julian System Date.
N	Y	N	N	N	SYS( 10)	String for Julian Day Number.
N	Y	N	N	N	SYS( 11)	Julian Day Number.
Y	Y	N	N	N	SYS( 2)	Seconds Since Midnight. Note that this performs the same task as SECONDS() except that SYS(2) will recognize if the system clock has been changed while SECONDS() won't.
N	Y	N	N	N	TIME()	Returns the current system time in 24-hour, 8-character string (hh:mm:ss) format.
N	N	Y	N	N	TTOC()	Converts a datetime expression to a Character value of a specified format.
N	N	Y	N	N	TTOD()	Returns a date value from a datetime expression.
N	N	Y	N	N	WEEK()	Returns a number representing the week of the year from a date or datetime expression.
N	Y	N	N	N	YEAR()	Returns the year from the specified date or datetime expression.

## DDE

Visual FoxPro supports DDE (Dynamic Data Exchange) to be compliant with older applications that aren't COM compliant, but you probably wouldn't be using them in new application development. Thus, I'll save a page or two by mentioning that they begin with "DDE."

Ob	In 2.x	In 3	In 5	In 6	Command/Function	Description
N	Y	N	N	N	DDE (MULTIPLE)	Functions to exchange data between VFP and other Microsoft Windows applications.

## Debugging

Debugging capabilities in Visual FoxPro are robust and flexible. I've included both the commands that work with all of the Debug and Trace windows as well as other tools used in the debugging process—the Coverage Analyzer and the ability to use assertions.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	_THROTTLE	Specifies execution speed of programs when the Trace window is open.
N	N	N	Y	N	ASSERT	Displays a message when an expression evaluates to .f.
N	N	Y	N	N	CLOSE DEBUGGER	Closes the debug window.
N	N	N	Y	N	DEBUG	Opens the debugger.
N	N	N	Y	N	DEBUGOUT	Drives command results to the Debug Output window.
N	N	N	Y	N	SET ASSERTS	Specifies if ASSERT commands are evaluated or ignored.
N	N	N	Y	N	SET COVERAGE	Turns code coverage on or off, or specifies a text file to which code coverage information is directed.
N	Y	N	N	N	SET DEBUG	Makes the Debug and Trace windows either available or unavailable from the Visual FoxPro menu system.
N	N	N	Y	N	SET DEBUGOUT	Directs debugging output to a file.
N	N	N	Y	N	SET EVENTLIST	Specifies events to track in the Debug Output Window or in a file specified with SET EVENTTRACKING.
N	N	N	Y	N	SET EVENTTRACKING	Turns event tracking on or off, or specifies a text file to which event tracking information is directed.
N	Y	N	N	N	SET STEP	Opens the Trace window and suspends program execution for debugging.
N	Y	N	N	N	SET TRBETWEEN	Enables or disables tracing between breakpoints in the Trace window.

## Environment – General

These commands and functions allow you to determine various miscellaneous attributes about the hardware of the system.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	N	Y	N	N	ISMOUSE()	Determines whether a mouse is present.
N	Y	N	N	N	OS()	Returns name and version of current OS.
N	Y	N	N	N	SET()	Returns the status of various SET commands.
N	Y	N	N	N	SET BELL	Turns the computer bell on or off and sets the bell attributes.
N	Y	N	N	N	SYS( 17)	Processor in use.
N	Y	N	N	N	SYSMETRIC()	Returns the size of the operating system's screen elements.

## Environment – Disk

These commands and functions allow you to determine various attributes about disk storage.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	DISKSPACE()	Returns the number of bytes available.
N	N	N	N	Y	DRIVETYPE()	Returns the type of drive (floppy, hard, and so on.)
N	Y	N	N	N	SYS(2020)	Default disk size.
N	Y	N	N	N	SYS(2022)	Disk cluster size.

## Environment – Keyboard

These functions allow you to determine various attributes about the keyboard.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	CAPSLOCK()	Toggles the CAPS LOCK key (even turns the light on and off!).
N	Y	N	N	N	FKMAX()	Returns the number of function keys.
N	Y	N	N	N	INSMODE()	Returns the value of the Insert key (and sets it).
N	Y	N	N	N	NUMLOCK()	Returns the value of the NumLock key (and sets it).

## Environment – Monitor

These functions allow you to determine various attributes about the monitor.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	SCOLS()	Returns the number of columns available in the main window.
N	Y	N	N	N	SROWS()	Returns the number of rows available in the main window.
N	Y	N	N	N	SYS(2006)	Current graphics card.

## Environment – Network

These functions allow you to determine various attributes about the network.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	N	N	N	Y	ANETRESOURCES()	Stuffs network shares and printers into an array; returns number of resources.
N	Y	N	N	N	ID()	Returns machine identification, including information about the current user.
N	Y	N	N	N	SYS( 0 )	Network Machine Info.

## Error handling

These commands and functions give you the ability to trap errors and collect information about the programming environment, such as the program name and line number. I've not included all the commands that grab environment information (such as what files are open or the status of memory variables).

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	N	Y	N	N	AERROR()	Stuffs error information into an array.
N	Y	N	N	N	ERROR()	Returns the error number triggered by an ON ERROR condition.
N	N	Y	N	N	ERROR	Generates a VFP error.
N	Y	N	N	N	LINENO()	Returns the line number of the program being executed.
N	Y	N	N	N	MESSAGE()	Returns the current message or contents of the bad program line.
N	Y	N	N	N	ON ERROR	Specifies a command that executes when an error occurs.
N	Y	N	N	Y	PROGRAM()	Returns the name of the current program.
N	Y	N	N	N	RETRY	Repeats the execution of a program line.
N	Y	N	N	N	SYS( 16)	Executing program file name.
N	Y	N	N	N	SYS(2018)	Error message parameter.

## File name manipulations

These commands and functions allow you to make a wide variety of manipulations to file names. Many of these were included in the FOXTOOLS library in previous versions but are now native to the language.



*Use a consistent naming scheme when referring to files. I've used the “DPFE” acronym for years when creating variable names for files. The “D” means the drive letter is included, “P” means the path (including the last backslash) is included, “F” means the file name (without the period or extension) is included, and “E” means the extension is included. For example:*

```
m.1cDPFEImport  E:\IMPORT\CUST.TXT
m.1cPFEImport   \IMPORT\CUST.TXT
m.1cFImport     CUST.TXT
m.1cFImport     CUST
m.1cDPIimport   E:\IMPORT\
```

The commands and functions in the following list allow you to move from one type of file to another easily. Using them in combination with a naming scheme makes short, foolproof work of using file names.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	N	N	N	Y	ADDBS()	Adds a backslash to a path if necessary.
N	N	N	N	Y	DEFAULTTEXT()	Returns a file name with a new extension.
N	N	N	N	Y	FORCEEXT()	Returns string with a new file extension.
N	N	N	N	Y	FORCEPATH()	Returns a file name with a new path.
N	N	N	N	Y	JUSTDRIVE()	Returns the drive letter from a complete path expression.
N	N	N	N	Y	JUSTEXT()	Returns the extension from a complete path expression.
N	N	N	N	Y	JUSTFNAME()	Returns the file name (with extension) from a complete path expression.
N	N	N	N	Y	JUSTPATH()	Returns the path from a complete path and file name expression.
N	N	N	N	Y	JUSTSTEM()	Returns the file name (without extension) from a complete path expression.
N	Y	N	N	N	SET FULLPATH	Specifies if CDX( ), DBF( ), MDX( ), and NDX( ) return the path in a file name.

## File information

These commands and functions provide a variety of status information about what and where files and directories are.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	ADIR()	Stuffs names of the contents of a directory into an array.
N	N	N	N	Y	AGETFILEVERSION()	Stuffs information about available Windows version resources into an array.
N	Y	N	N	N	CURDIR()	Returns the current directory (but not the drive).
N	Y	N	N	N	DIR	Similar to the DOS DIR command.
N	N	N	Y	N	DIRECTORY()	Returns true if a directory exists.
N	Y	N	N	N	DISPLAY FILES	Displays information about files.
N	N	Y	N	N	FDATE()	Returns the date/time that a file was last modified.
N	Y	N	N	N	FILE()	Returns true if file is found.
N	N	Y	N	N	FTIME()	Returns the time that a file was last modified.
N	Y	N	N	N	FULLPATH()	Returns a path relative to a file.
N	Y	N	N	N	GETENV()	Returns MS-DOS environment variables.
N	Y	N	N	N	HOME()	Returns the name of the Visual FoxPro (and VS) installation directory.
N	Y	N	N	N	LUPDATE()	Returns the date a file was last modified.
N	Y	N	N	N	SET DEFAULT	Specifies the default drive and directory.
N	Y	N	N	N	SYS(2000)	File name wildcard match.
N	Y	N	N	N	SYS(2014)	Minimum path.
N	Y	N	N	N	SYS( 5)	Default drive.

## File selection

These commands and functions provide a standard interface for letting the user select a file.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	GETDIR()	Displays the Select Directory dialog.
N	Y	N	N	N	GETFILE()	Displays the File Open dialog.
N	N	Y	N	N	GETPICT()	Displays the Open Picture dialog.
N	N	N	Y	N	LOADPICTURE()	Creates an object reference to an image file.
N	Y	N	N	N	LOCFILE()	Locates a file on disk and displays a dialog.
N	Y	N	N	N	PUTFILE()	Displays the Save As dialog.

## File and directory writing

These commands and functions all have something to do with writing files to disk, renaming files, or deleting files.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	N	Y	N	N	CD/CHDIR	Same as DOS CD.
N	Y	N	N	N	CLOSE ALTERNATE	Closes an alternate file.
N	Y	N	N	N	COPY FILE	Copies a file.
N	Y	N	N	N	COPY MEMO	Copies contents of a memo field to a text file.
N	Y	N	N	N	COPY TO	Creates a new file from the contents of a table.
N	Y	N	N	N	DELETE FILE	Deletes a file from disk.
N	Y	N	N	N	ERASE	Deletes a file from disk.
N	Y	N	N	N	EXPORT	Copies data from a table to a file.
N	N	N	N	Y	FILETOSTR()	Returns the contents of a file as a string.
N	Y	N	N	N	IMPORT	Imports data from a non-.DBF file format.
N	N	Y	N	N	MD	Creates a directory.
N	N	Y	N	N	RD	Deletes a directory.
N	Y	N	N	N	RENAME	Renames a file.
N	Y	N	N	N	SAVE TO	Saves current memory variables.
N	N	N	Y	N	SAVEPICTURE()	Creates a bitmap file from a picture object reference.
N	Y	N	N	N	SET ALTERNATE	Directs screen or printer output created with ?, ??, DISPLAY, or LIST to a text file.
N	N	N	N	Y	STRTOFILE()	Writes the contents of a character string to a file.
N	Y	N	N	N	TYPE	Displays the contents of a file.

## Fonts

These functions work with fonts currently installed on the system.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	AFONT()	Stuffs available font information into an array.
N	Y	N	N	N	FONTMETRIC()	Returns font attributes.
N	Y	N	N	N	GETFONT()	Displays the Font dialog.
N	Y	N	N	N	TXTWIDTH()	Returns the length of a character expression with respect to the average character width for a font.

## Help

These commands and functions provide you with capabilities to develop and include Help files with your application.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	HELP	"Ummm, duh!"
N	Y	N	N	N	SET HELP	Enables or disables Microsoft Visual FoxPro Online Help or specifies a Help file.
N	Y	N	N	N	SET HELPFILTER	Enables Visual FoxPro to display a subset of .DBF-style help topics in the Help window.
N	Y	N	N	N	SET TOPIC	Specifies the Help topic or topics to open when you invoke the Visual FoxPro Help system.
N	N	Y	N	N	SET TOPIC ID	Specifies the Help topic to display when you invoke the Visual FoxPro Help system. The Help topic is based on the topic's context ID.
N	N	Y	N	N	SYS(1023)	Enable Help Diagnostic Mode.
N	N	Y	N	N	SYS(1024)	Disable Help Diagnostic Mode.

## Interrupts (Mouse, Keyboard, Other)

These commands and functions act as interrupts to the system. Most of the time they take the form of keyboard or mouse actions, but there are other cases as well.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	_DBLCLICK	Specifies the time interval between single and double mouse clicks.
N	N	N	N	Y	AMOUSEOBJ()	Stuffs mouse-pointer and mouse-object hover information into an array.
N	Y	N	N	N	CHRSAW()	Determines if the keyboard buffer contains a character.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	CLEAR TYPEAHEAD	Clears the keyboard buffer.
N	Y	N	N	N	FKLABEL()	Returns the label of the function key from the number.
N	Y	N	N	N	INKEY()	Returns a numeric value corresponding to a mouse click or key press.
N	Y	N	N	N	KEYBOARD	Stuffs a character expression into the keyboard buffer.
N	Y	N	N	N	LASTKEY()	Returns a number matching the last key pressed.
Y	Y	N	N	N	MDOWN()	Tells you whether a mouse button is down at the moment.
N	N	Y	N	N	MOUSE	Performs mouse operations.
N	Y	N	N	N	ON()	Returns the command assigned to various ON commands.
N	Y	N	N	N	ON ESCAPE	Specifies a command that executes when the ESC key is pressed.
N	Y	N	N	N	ON KEY LABEL	Specifies a command that executes when a key is pressed or the mouse is clicked.
N	Y	N	N	N	ON SHUTDOWN	Specifies a command that executes when exiting VFP or Windows.
N	Y	N	N	N	POP KEY	Restores ON KEY LABEL assignments.
N	Y	N	N	N	PUSH KEY	Places all current ON KEY LABEL settings on a stack.
N	Y	N	N	N	SET ESCAPE	Determines whether pressing the ESC key interrupts program and command execution.
N	Y	N	N	N	SET FUNCTION	Assigns an expression (keyboard macro) to a function key or key combination.
N	Y	N	N	N	SET KEYCOMP	Controls Visual FoxPro keystroke navigation.
N	Y	N	N	N	SET TYPEAHEAD	Specifies the maximum number of characters that can be stored in the type-ahead buffer.

## International

These functions and commands can help with international issues, including code pages and settings of delimiters that vary from country to country.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	CP CONVERT()	Converts character expressions to a different code page.
N	Y	N	N	N	CPCURRENT()	Returns the current code page setting.
N	Y	N	N	N	CPDBF()	Returns the code page setting of a table.
N	N	Y	N	N	GETCP()	Displays the Code Page dialog.
N	N	Y	N	N	IMESTATUS()	Turns the IME window on or off.

Ob	In 2.x	In 3	In 5	In 6	Command/Function	Description
N	N	Y	N	N	ISLEADBYTE()	Determines whether the first character in an expression is the lead byte of a double-byte character.
N	N	N	N	Y	SET BROWSEIME	Specifies if the Input Method Editor is opened when you navigate to a text box in a Browse window.
N	Y	N	N	N	SET CPCCOMPILE	Specifies the code page for compiled programs.
N	Y	N	N	N	SET CPDIALOG	Specifies whether the Code Page dialog box is displayed when a table is opened.
N	Y	N	N	N	SET CURRENCY	Defines the currency symbol and specifies its position in the display of numeric, currency, float, and double expressions.
N	Y	N	N	N	SET POINT	Determines the decimal point character used in the display of numeric and currency expressions.
N	Y	N	N	N	SET SEPARATOR	Controls the character used to separate groups of digits.
N	N	Y	N	N	SET SYSFORMATS	Specifies whether Visual FoxPro for Windows system settings are updated with the current Microsoft Windows system settings.
N	Y	N	N	N	SYS( 15)	Character translation.
N	Y	N	N	N	SYS( 20)	Transform German text.

## Low-level file functions

There are three types of capabilities represented by the items in this list. The BIT-wise operators—BITNOT, BITSHIFT, etc.—allow you to work with individual bits of a byte. The low-level file functions, such as FWRITE and FSEEK, allow you to create and manipulate a file on a byte-by-byte basis. And all of the tools used to access and manipulate the Windows API, such as DECLARE DLL and CTOBIN, are in this list as well.

Ob	In 2.x	In 3	In 5	In 6	Command/Function	Description
N	N	N	Y	N	BINTOC()	Integer to binary conversion.
N	N	Y	N	N	BITAND()	Bitwise AND.
N	N	Y	N	N	BITCLEAR()	Bitwise Clear.
N	N	Y	N	N	BITLSHIFT()	Bitwise Left Shift.
N	N	Y	N	N	BITNOT()	Bitwise NOT.
N	N	Y	N	N	BITOR()	Bitwise OR.
N	N	Y	N	N	BITRSHIFT()	Bitwise Right Shift.
N	N	Y	N	N	BITSET()	Bitset to a 1.
N	N	Y	N	N	BITTEST()	Tests if a bit is 1.
N	N	Y	N	N	BITXOR()	Bitwise XOR.
N	N	Y	N	N	CLEAR DLLS	Clears all external shared libraries.
N	N	N	Y	N	CTOBIN()	Converts a binary character string to an integer value.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	N	Y	N	N	DECLARE DLL	Registers a function in an external library (.DLL).
N	N	Y	N	N	DISPLAY DLLS	Displays information about functions registered with DECLARE.
N	Y	N	N	N	FCHSIZE()	Low-level file function: Changes the size of a file.
N	Y	N	N	N	FCLOSE()	Low-level file function: Flush and close.
N	Y	N	N	N	FCREATE()	Low-level file function: Creates and opens.
N	Y	N	N	N	FEOF()	Low-level file function: End of file flag.
N	Y	N	N	N	FERROR()	Low-level file function: Error number.
N	Y	N	N	N	FFLUSH()	Low-level file function: Flushes.
N	Y	N	N	N	FGETS()	Low-level file function: Returns a series of bytes.
N	Y	N	N	N	FOPEN()	Low-level file function: Opens a file.
N	Y	N	N	N	FPUTS()	Low-level file function: Writes data.
N	Y	N	N	N	FREAD()	Low-level file function: Returns a number of bytes.
N	Y	N	N	N	FSEEK()	Low-level file function: Moves pointer.
N	Y	N	N	N	FSIZE()	Low-level file function: Determines size.
N	Y	N	N	N	FWRITE()	Low-level file function: Writes a string.
N	Y	N	N	N	RELEASE LIBRARY	Removes an API library from memory.
N	Y	N	N	N	SET LIBRARY	Opens an external API (application programming interface) library file.
N	N	N	N	Y	SYS(3056)	Read Registry Settings.
N	Y	N	N	N	RUN	Executes an external program.

## Macros

These commands read and play macros.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	CLEAR MACROS	Clears keyboard macros.
N	Y	N	N	N	PLAY MACRO	Executes a keyboard macro.
N	Y	N	N	N	RESTORE MACROS	Restores keyboard macros.
N	Y	N	N	N	SAVE MACROS	Saves a set of macros.
N	Y	N	N	N	SET MACKEY	Specifies a key or key combination that displays the Macro Key Definition dialog box.

## Math

This area is a catch-all for the functions and commands that do some sort of math. There are the trigonometric and exponential functions, such as COS, DTOR and LOG10 (I know, many of you are wondering why I didn't move these to front of the book or otherwise emphasize them more heavily); financial functions, such as PAYMENT; numeric functions, such as ROUND; and statistical functions, such as AVERAGE.

## Math – Financial

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	FV()	Future value of an investment.
N	N	Y	N	N	MTON()	Converts a currency ("money") expression to a numeric one.
N	N	Y	N	N	NTOM()	Converts a numeric expression to a currency ("money") one.
N	Y	N	N	N	PAYMENT()	Returns the amount of each period payment on a fixed-interest loan.
N	Y	N	N	N	PV()	Present value.

## Math – Numeric

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	%	Modulus operator: Returns the remainder.
N	Y	N	N	N	ABS()	Absolute value.
N	Y	N	N	N	CEILING()	Rounds a non-integral number to the next integer.
N	Y	N	N	N	FLOOR()	Returns the integer "less or equal to" value.
N	Y	N	N	N	INT()	Returns the integral portion of a number.
N	Y	N	N	N	MAX()	Returns the largest of a set of values.
N	Y	N	N	N	MIN()	Returns the smallest of a set of values.
N	Y	N	N	N	MOD()	Mathematical modulus.
N	Y	N	N	N	RAND()	Random number generator.
N	Y	N	N	N	ROUND()	Rounds a numeric expression.
N	Y	N	N	N	SET DECIMALS	Specifies the number of decimal places displayed in numeric expressions.
N	Y	N	N	N	SET FIXED	Specifies if the number of decimal places used in the display of numeric data is fixed.
N	Y	N	N	N	SIGN()	Returns a numeric flag based on the sign of an expression.
N	Y	N	N	N	SQRT()	Square root.

## Math – Statistical

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	AVERAGE	Returns the average from a table.
N	Y	N	N	N	CALCULATE	Returns a statistical operation (MAX, MIN, etc.) on a table.
N	Y	N	N	N	COUNT	Returns the number of records that meet certain criteria.
N	Y	N	N	N	SUM	Totals all or specified numeric fields in the currently selected table.
N	Y	N	N	N	TOTAL	Computes totals for numeric fields in the currently selected table.

## Math – Trig and Exponential

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	ACOS()	Arc cosine.
N	Y	N	N	N	ASIN()	Arc sin.
N	Y	N	N	N	ATAN()	Arc tangent.
N	Y	N	N	N	ATN2()	Arc tangent in all four quadrants.
N	Y	N	N	N	COS()	Returns the cosine of a numeric expression.
N	Y	N	N	N	DTOR()	Converts degrees to radians.
N	Y	N	N	N	EXP()	Returns "e" raised to a power of "x".
N	Y	N	N	N	LOG()	Returns the natural log of an expression.
N	Y	N	N	N	LOG10()	Returns the common log of an expression.
N	Y	N	N	N	PI()	The number pi.
N	Y	N	N	N	RTOD()	Converts radians to degrees.
N	Y	N	N	N	SIN()	Returns the sine of an angle.
N	Y	N	N	N	TAN()	Returns the tangent of an angle.

## Memory

These commands and functions have to do with memory. Now that home computers are being sold with 128 MB of RAM, memory isn't the precious commodity it once was, and you might develop for a long time without using any of these.

DISPLAY MEMORY does display the memory variables currently defined, which can occasionally be handy.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	CLEAR ALL	Releases all but system memory variables and compiled program buffer.
N	Y	N	N	N	CLEAR MEMORY	Clears all memvars.
N	N	Y	N	N	CLEAR RESOURCES	Clears image cache.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	DISPLAY MEMORY	Displays memory variable contents.
N	N	N	Y	N	SYS(3050)	Sets buffer memory size.
N	Y	N	N	N	SYS(1016)	How much memory is currently occupied by user-defined objects.

## Miscellaneous

Some commands and functions just don't fit anywhere else. That's why the word "miscellaneous" is in the dictionary, and in this book.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	_CALCMEM	Contains the numeric value that Microsoft Visual FoxPro stores in the Calculator's memory.
N	Y	N	N	N	_CALCVALUE	Contains the numeric value that the Calculator displays.
N	Y	N	N	N	_CLIPTEXT	Contains the contents of the Clipboard.
N	Y	N	N	N	_DIARYDATE	Contains the current date in the Calendar/Diary.
N	Y	N	N	N	DIFFERENCE()	Returns a value that shows the relative phonetic difference between two expressions.
N	Y	N	N	N	SET COMPATIBLE	Controls compatibility with Microsoft FoxBASE+ and other Xbase languages.
N	Y	N	N	N	SOUNDEX()	Returns a phonetic representation of an expression.
N	Y	N	N	N	SYS(2007)	Checksum Value.
N	Y	N	N	N	SYS(2015)	Unique Procedure Name.

## OOP

These commands and functions allow you to work with classes and objects.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
Y	N	Y	N	N	::	Runs a parent class method from within a subclass method.
N	N	Y	N	N	ACLASS()	Stuffs an object's class name and hierarchy into an array.
N	N	Y	N	N	ADD CLASS	Adds a class definition to a class library.
N	N	Y	N	N	ADD OBJECT	Defines a new class based on an existing one.
N	N	N	N	Y	AGETCLASS()	Open dialog for selecting class; stuffs selection into an array.
N	N	Y	N	N	AINSTANCE()	Places instance information into an array.
N	N	Y	N	N	AMEMBERS()	Stuffs names of object members into an array.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	N	Y	N	N	ASELOBJ()	Places object references to selected controls into an array.
N	N	N	N	Y	AVCXCLASSES()	Stuffs class information from a class library into an array.
N	N	Y	N	N	CLEAR CLASS	Clears class definition.
N	N	Y	N	N	CLEAR CLASSLIB	Clears all class libraries from memory.
N	N	Y	N	N	COMPOBJ()	Returns .t. if properties and property values of two objects are the same.
N	N	Y	N	N	CREATE CLASS	Creates a class definition.
N	N	Y	N	N	CREATE CLASSLIB	Creates a class library.
N	N	Y	N	N	CREATEOBJECT()	Creates an object from a class definition or Automation application.
N	N	Y	N	N	DEFINE CLASS	Creates a class.
N	N	Y	N	N	DISPLAY OBJECTS	Displays information about objects.
N	N	N	Y	N	DODEFAULT()	Executes a parent class of the same name.
N	N	N	Y	N	GETPEM()	Returns the current value for a property or program code for an event at design time.
N	N	N	N	Y	NEWOBJECT()	Creates a new object or class from a visual class definition.
N	N	Y	N	N	RELEASE CLASSLIB	Closes open class libraries.
N	N	Y	N	N	REMOVE CLASS	Removes a class definition from a class library.
N	N	Y	N	N	RENAME CLASS	Renames a class definition in a class library.
N	N	Y	N	N	SET CLASSLIB	Opens a .VCX visual class library containing class definitions.
N	N	Y	N	N	SYS(1270)	Object reference to the object under the mouse.
N	N	Y	N	N	SYS(1271)	Object's SCX File.
N	N	Y	N	N	SYS(1272)	Object Hierarchy.

## OOP – Control-specific

Thrown in here are a few commands and functions that are more geared toward working with a form and its controls (although they do work with objects in many cases).

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	N	N	N	Y	_INCLUDE	Specifies a default header file included with user-defined classes, forms, or form sets.
N	N	Y	N	Y	CREATE FORM	Creates a form.
N	N	Y	N	N	OBJTOCLIENT()	Returns the position of a control with respect to its form.
N	N	Y	N	N	PEMSTATUS()	Returns an attribute for a property, event, method or object.
N	N	Y	N	N	SYS(1269)	Property Information.

## Printing

These all have to do with creating reports and formatting output or dealing with printer hardware.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	???	Sends output to the printer.
N	N	Y	N	N	_ASCIICOLS	Specifies the number of columns in a text file created with REPORT ... TO FILE ASCII.
N	N	Y	N	N	_ASCIIROWS	Specifies the number of rows in a text file created with REPORT ... TO FILE ASCII.
N	Y	N	N	N	_PAGENO	Contains the current page number.
N	N	Y	N	N	APRINTERS()	Stuffs names of Windows printers into an array.
N	Y	N	N	N	CREATE LABEL	Creates a label.
N	Y	N	N	N	CREATE REPORT	Creates a report.
N	Y	N	N	N	EJECT	Sends a formfeed to the printer.
N	Y	N	N	N	EJECT PAGE	Sends a conditional page advance to the printer.
N	N	Y	N	N	GETPRINTER()	Displays the Print dialog.
N	Y	N	N	N	LABEL	Prints labels.
N	Y	N	N	N	ON PAGE	Specifies a command that is executed when printed output hits a specified line number.
N	Y	N	N	N	PCOL()	Returns the printer's column position.
N	Y	N	N	N	PROW()	Returns the printer's column position.
N	Y	N	N	N	PRTINFO()	Returns the current specified printer setting.
N	Y	N	N	Y	REPORT	Runs a report.
N	Y	N	N	N	SET HEADINGS	Determines whether column headings are displayed for fields and whether file information is included when TYPE is issued to display the contents of a file.
N	Y	N	N	N	SET PRINTER	Enables or disables output to the printer, or routes output to a file, port, or network printer.
N	Y	N	N	N	SET SPACE	Determines whether a space is displayed between fields or expressions when you use the ? or ?? command.
N	Y	Y	N	N	SYS(1037)	Page Setup dialog box.

## Programming – Comments

These have to do with commenting your programs.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	&&	Comment character (particularly used for in-line comments).
N	Y	N	N	N	*	Comment character.
N	Y	N	N	N	NOTE	Marks a line of program code as a comment.

## Program control structures

You would typically use these commands and functions only in a program—not in the Command window—to control program flow and provide logic branches. I've included the # commands as well.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	#DEFINE	Creates/releases compile-time constants.
N	Y	N	N	N	#IF	Conditionally includes source code at compile time.
N	N	Y	N	N	#IFDEF	Conditionally includes a set of commands at compile time if a compile-time constant is included.
N	N	Y	N	N	#INCLUDE	Tells the Visual FoxPro preprocessor to treat the contents of a specified header file as if it appeared in a VFP program.
N	Y	N	N	N	DO CASE CASE ENDCASE	Conditional logic structure—runs first group of commands grouped under a valid CASE.
N	Y	N	N	N	DO WHILE ENDDO	Conditional logic structure—repeats loop until condition is false.
N	Y	N	N	N	EXIT	Exits a DO WHILE, SCAN or FOR logic structure.
N	Y	N	N	N	FOR ENDFOR/NEXT	Looping construct that executes a batch of commands.
N	N	N	Y	N	FOR EACH ENDFOR/NEXT	Executes a batch of commands for each element in an array or collection.
N	Y	N	N	N	IF ELSE ENDIF	Logical construct that conditionally executes a batch of commands.
N	Y	N	N	N	IIF()	Returns one of two values based on a logical expression's evaluation.
N	Y	N	N	N	LOOP	Ends the current pass in a loop.
N	Y	N	N	N	OTHERWISE	Provides a mechanism for trapping conditions when no CASE statements evaluate to True.
N	Y	N	N	N	SCAN ENDSCAN	Moves record pointer through a group of records and executes a batch of commands for each record.
N	N	Y	N	N	WITH ENDWITH	Specifies an object reference.

## Program event handling

These commands and functions are used to put Visual FoxPro into a wait state to deal with events, or otherwise work in an event-driven mode.

Ob	In 2.x	In 3	In 5	In 6	Command/Function	Description
N	N	Y	N	N	CLEAR EVENTS	Clears stack of pending events.
N	N	Y	N	N	DOEVENTS()	Executes pending events.
N	N	Y	N	N	READ EVENTS	Starts event processing.

## Program subroutines

These commands and functions are used to work with subroutines and program termination.

Ob	In 2.x	In 3	In 5	In 6	Command/Function	Description
N	Y	N	N	N	&	Macro substitution (for commands).
N	Y	N	N	N	CANCEL	Terminates execution of a program.
N	Y	N	N	N	CLEAR PROGRAM	Clears the compiled program buffer.
N	Y	N	N	N	CLOSE PROCEDURE	Closes a procedure file.
N	N	Y	N	N	DISPLAY PROCEDURES	Displays names of procedures in the current database.
N	Y	N	N	N	DO	Executes a program.
N	Y	N	N	N	FUNCTION	Identifies the beginning of a user-defined function.
N	Y	N	N	N	NORMALIZE()	Converts a character expression so it can be compared with VFP function return values.
N	Y	N	N	N	PROCEDURE	Identifies the beginning of a user-defined procedure.
N	Y	N	N	N	QUIT	Quits Visual FoxPro.
N	N	Y	N	N	RELEASE PROCEDURE	Closes open procedure files.
N	Y	N	N	N	RESUME	Continues execution of a program that was suspended.
N	Y	N	N	N	RETURN	Returns from a subroutine to a calling program.
N	Y	N	N	N	SET DEVELOPMENT	Causes Visual FoxPro to compare the creation date and time of a program with those of its compiled object file when the program is run.
N	Y	N	N	N	SET LOGERRORS	Determines whether Visual FoxPro sends compilation error messages to a text file.
N	Y	N	N	N	SET PROCEDURE	Opens a procedure file.
N	Y	N	N	N	SUSPEND	Pauses program execution.

## Program variable scoping

These commands and functions are used to work with variable scoping and passing parameters to and from subroutines.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	N	Y	N	N	LOCAL	Scopes a variable to the current procedure.
N	N	Y	N	N	LPARAMETERS	Accepts data from a calling program and makes the variables local.
N	Y	N	N	N	PARAMETERS	Accepts data from a calling program and makes the variables private.
N	Y	N	N	N	PCOUNT()	Returns the number of parameters passed to the current program.
N	Y	N	N	N	PRIVATE	Creates private memory variables.
N	Y	N	N	N	PUBLIC	Creates a global variable.
N	Y	N	N	N	SET UDFPARMS	Specifies if Microsoft Visual FoxPro passes parameters to a user-defined function (UDF) by value or by reference.

## Project manager

These commands compile source code in design surfaces (COMPILE FORM) and build files (BUILD APP).

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	BUILD APP	Builds an .APP file from a project.
N	N	N	Y	N	BUILD DLL	Builds a .DLL file from a project.
N	Y	N	N	N	BUILD EXE	Builds an .EXE file from a project.
N	Y	N	N	N	BUILD PROJECT	Compiles (or recompiles) files in a project.
N	Y	N	N	N	COMPILE	Compiles source file(s) and creates object file(s).
N	N	N	Y	N	COMPILE CLASSLIB	Compiles a visual class library.
N	N	Y	N	Y	COMPILE DATABASE	Compiles stored procedures in a VFP database.
N	N	Y	N	N	COMPILE FORM	Compiles a form.
N	Y	N	N	N	COMPILE LABEL	Compiles a label.
N	Y	N	N	N	COMPILE REPORT	Compiles a report.
N	Y	N	N	N	CREATE PROJECT	Creates a project.

## String functions

These commands and functions cover a variety of string operations, including finding pieces in the middle of a string, working with substrings, and converting to and from strings with other data types.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	\$	"Is contained in" (returns .t. or .f.).
N	Y	N	N	N	_MLINE	Contains the line offset for the MLINE( ) function.
N	N	N	N	Y	ALINES()	Stuffs lines from a string into an array.
N	Y	N	N	N	ALLTRIM()	Removes leading and trailing spaces from a character expression.
N	Y	N	N	N	ASC()	Returns ASCII value for first character of a string.
N	Y	N	N	N	AT()	Returns the position of the first character of an expression within another character expression.
N	N	Y	N	N	AT_C()	AT for double-byte.
N	Y	N	N	N	ATC()	AT without regard for case.
N	N	Y	N	N	ATCC()	ATC for double-byte.
N	Y	N	N	N	ATCLINE()	ATLINE for double-byte.
N	Y	N	N	N	ATLINE()	Returns line number of the first character of an expression within another character expression.
N	Y	N	N	N	CHR()	Returns a character associated with an ASCII value.
N	Y	N	N	N	CHRTRAN()	Stuffs a character expression in the place of another.
N	N	Y	N	N	CHRTRANC()	CHRTRAN double-byte.
N	Y	N	N	N	ISALPHA()	Determines if the first character in an expression is alphabetic.
N	Y	N	N	N	ISDIGIT()	Determines whether the first character in an expression is a number.
N	Y	N	N	N	ISLOWER()	Determines whether the first character of an expression is lowercase.
N	Y	N	N	N	ISUPPER()	Determines whether the first character of an expression is uppercase.
N	Y	N	N	N	LEFT()	Returns the N left-most characters from an expression.
N	N	Y	N	N	LEFTC()	LEFT for double-byte.
N	Y	N	N	N	LEN()	Returns the number of characters in an expression.
N	N	Y	N	N	LEN()	LEN for double-byte.
N	Y	N	N	N	LIKE()	Compares two character expressions.
N	N	Y	N	N	LIKEC()	LIKE for double-byte.
N	Y	N	N	N	LOWER()	Converts an expression to all lowercase.
N	Y	N	N	N	LTRIM()	Removes leading blanks from a character expression.
N	Y	N	N	N	MEMLINES()	Returns the number of lines in a string.
N	Y	N	N	N	MLINE()	Returns a specific line from a string.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	OCCURS()	Returns how many times an expression is contained in another expression.
N	Y	N	N	N	PADL/R/C()	Adds characters to a character string to pad it to a specified length.
N	Y	N	N	N	PROPER()	Converts an expression to all proper case.
N	Y	N	N	N	RAT()	Returns the numeric position of the rightmost occurrence of a character string.
N	N	Y	N	N	RATC()	RAT for double-byte.
N	Y	N	N	N	RATLINE()	Returns the line number of the last occurrence of a character string, starting with the last line.
N	Y	N	N	N	REPLICATE()	Returns a character string that repeats a second string a specified number of times.
N	Y	N	N	N	RIGHT()	Returns the N right-most characters from an expression.
N	N	Y	N	N	RIGHTC()	RIGHT for double-byte.
N	Y	N	N	N	RTRIM()	Removes trailing blanks from a character expression.
N	Y	N	N	N	SET MEMOWIDTH	Specifies the displayed width of memo fields and character expressions.
N	Y	N	N	N	SPACE()	Returns a string of spaces.
N	Y	N	N	N	STR()	Converts a numeric expression to a character string.
N	N	Y	N	N	STRCONV()	Converts character expressions from one locale-specific form to another.
N	Y	N	N	N	STRTRAN()	Looks for a character expression for occurrences of a second expression and replaces them with a third.
N	Y	N	N	N	STUFF()	Stuffs a character string into another character string, optionally replacing an expression.
N	N	Y	N	N	STUFFC()	STUFF for double-byte.
N	Y	N	N	N	SUBSTR()	Returns a substring from an expression.
N	N	Y	N	N	SUBSTRC()	SUBSTR for double-byte.
N	Y	N	N	Y	TRANSFORM()	Returns a character string from an expression in a format determined by a format code.
N	Y	N	N	N	TRIM()	Returns the specified character expression with all trailing blanks removed.
N	Y	N	N	N	UPPER()	Converts an expression to all uppercase.
N	Y	N	N	N	VAL()	Returns a numeric value from a character expression composed of numbers.

## Text merge

These commands perform text-merge operations.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	\ and \\	Outputs (prints or displays) lines of text.
N	Y	N	N	N	_PRETEXT	Specifies a character expression to preface text-merge lines.
N	Y	N	N	N	_TEXT	Directs output from the \   \\ and TEXT ... ENDTEXT text-merge commands to a low-level file.
N	Y	N	N	N	SET TEXTMERGE	Enables or disables the evaluation of fields, variables, array elements, functions, or expressions that are surrounded by text-merge delimiters, and lets you specify text-merge output.
N	Y	N	N	N	SET TEXTMERGE DELIMITERS	Specifies the text-merge delimiters.
N	Y	N	N	N	TEXT ENDTEXT	Outputs lines of text, the results of expressions and functions, and the contents of variables.

## User interface

These aren't involved in creating forms or menus; rather, they're used for getting VFP to communicate system information to the user, such as status bars, progress meters, and so on.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	CLEAR	Clears the active window.
N	Y	N	N	N	DISPLAY	Displays contents of the current table.
N	Y	N	N	N	DISPLAY STATUS	Displays information about the environment.
N	N	Y	N	N	DO FORM	Executes a form.
Y	Y	N	N	N	GETCOLOR()	Displays the Windows Color Picker dialog.
N	Y	N	N	N	LIST	Similar to DISPLAY without pauses.
N	N	Y	N	N	MESSAGEBOX()	Displays a message box with user prompts.
N	Y	N	N	N	SET CONSOLE	Enables or disables output to the main Visual FoxPro window or to the active user-defined window from within programs.
N	Y	N	N	N	SET CURSOR	Determines whether the insertion point is displayed when Visual FoxPro waits for input.
N	Y	N	N	N	SET MARK TO	Specifies a delimiter for the display of date expressions.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	SET MESSAGE	Defines a message for display in the main Visual FoxPro window or in the graphical status bar, or specifies the location of messages for user-defined menu bars and menu commands.
N	Y	N	N	N	SET NOTIFY	Enables or disables the display of certain system messages.
N	N	Y	N	N	SET NULLDISPLAY	Specifies the text displayed for null values.
N	Y	N	N	N	SET ODOMETER	Specifies the reporting interval of the record counter for commands that process records.
N	Y	N	N	N	SET STATUS	Displays or removes the character-based status bar.
N	Y	N	N	N	SET STATUS BAR	Displays or removes the graphical status bar.
N	Y	N	N	N	SET TALK	Determines whether Visual FoxPro displays command results.
N	Y	N	N	N	WAIT	Displays a message and pauses VFP execution until the user presses a key or clicks the mouse.

## Variables

A miscellany of functions that operate on variables.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	? and ??	Evaluates expressions and displays the results.
N	Y	N	N	N	BETWEEN()	Returns .t. if a value is inclusively between two other values.
N	Y	N	N	N	COPY TO ARRAY	Creates an array from selected data in a table.
N	Y	N	N	N	EMPTY()	Determines whether an expression is empty.
N	Y	N	N	N	EVALUATE()	Evaluates a character expression and returns the result.
N	Y	N	N	N	GETEXPR	Displays the Expression Builder dialog.
N	Y	N	N	N	INLIST()	Determines whether an expression is contained in a list of expressions.
N	Y	N	N	N	ISBLANK()	Determines whether an expression is blank.
N	N	Y	N	N	ISNULL()	Determines whether an expression is .NULL.
N	N	Y	N	N	NVL()	Returns a non-null value from two expressions.
N	Y	N	N	N	RELEASE	Removes variables from memory.
N	Y	N	N	N	REPLACE FROM ARRAY	Updates fields in a table with values from an array.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	RESTORE FROM	Retrieves variables saved in a file or memo field.
N	Y	N	N	N	SET EXACT	Specifies the rules Visual FoxPro uses when comparing two strings of different lengths.
N	Y	N	N	N	STORE	Stores data to a variable.
N	Y	N	N	N	TYPE()	Evaluates an expression and returns the data type of its contents.
N	N	N	N	Y	VARTYPE()	Returns the data type of an expression.

## VFP

These commands and functions are used by VFP while VFP is running. Some of them are system variables that describe hooks to identify an outside program, such as the Class Browser, the Expression Builder, or the Menu Generator. Others have to do with how VFP interacts with the environment, including startup and configuration options, the splash screen, and the resource file.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	_ASSIST	Specifies the program that is automatically loaded during startup.
N	Y	N	N	N	_BEAUTIFY	Specifies the beautification application for Visual FoxPro programs. Runs when you select the Beautify command from the Tools menu.
N	N	Y	N	N	_BROWSER	Contains the name of the class browser application.
N	N	Y	N	N	_BUILDER	Contains the name of the Visual FoxPro builder application.
N	N	Y	N	N	_CONVERTER	Contains the name of the Microsoft Visual FoxPro converter application.
N	N	N	Y	N	_COVERAGE	Contains the name of the Visual FoxPro application that creates the Debugger coverage and profiler output.
N	N	N	N	Y	_GALLERY	Specifies the program that is executed when you select Component Gallery from the Tools menu.
N	N	N	N	Y	_GENHTML	Specifies an HTML-generation program.
N	Y	N	N	N	_GENMENU	Specifies a menu-generation program.
N	Y	N	N	N	_GENXTAB	Specifies the program used by the Query Designer when the CrossTab check box is checked.
N	N	N	N	Y	_GETEXPR	Specifies the program that is executed when you issue the GETEXPR command or invoke the Expression Builder dialog box from within Visual FoxPro.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	N	N	N	Y	_SAMPLES	Contains the path of the directory in which the Microsoft Visual FoxPro samples are installed.
N	N	N	Y	N	_SCCTEXT	Specifies a Visual FoxPro conversion program that handles translating Visual FoxPro binary files into text equivalents and back.
N	N	Y	N	N	_SCREEN	Specifies properties and methods for the main Visual FoxPro window.
N	Y	N	N	N	_SHELL	Specifies a program shell.
N	Y	N	N	N	_SPELLCHK	Specifies a spell-check program for the Visual FoxPro text editor.
N	Y	N	N	N	_STARTUP	Specifies the name of the application that runs when you start Visual FoxPro.
N	N	N	Y	N	_VFP	References the Application object for the current instance of Visual FoxPro.
N	N	Y	N	N	_WIZARD	Contains the name of the Visual FoxPro wizard application.
N	Y	N	N	N	ASSIST	Automatically loads the program identified by the _ASSIST variable.
N	Y	N	N	N	SET PATH	Specifies a path for file searches.
N	Y	N	N	N	SET RESOURCE	Updates or specifies a resource file.
N	Y	N	N	N	SYS(2010)	Config.sys file settings.
N	Y	N	N	N	SYS(2019)	Configuration file name/location.
N	Y	N	N	N	SYS(2023)	Temporary path for VFP Files.
N	Y	N	N	N	SYS( 9)	Visual FoxPro serial number.
N	Y	N	N	Y	VERSION()	Returns information about the Visual FoxPro version you are using.

## Visual Studio

These commands and functions have to do with how Visual FoxPro works with Visual Studio components such as ActiveDocs, OLE, and COM components.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	N	N	N	Y	_RUNACTIVEDOC	Specifies an application that launches an Active Document.
N	N	N	N	Y	COMARRAY()	Determines how arrays are passed to COM objects.
N	N	N	N	Y	COMCLASSINFO()	Returns registry information about a COM object.
N	N	N	N	Y	COMRETURNERROR()	Populates the COM exception structure.
N	N	N	Y	N	CREATEBINARY()	Creates a binary character string from a character string.
N	N	N	N	Y	CREATEOBJECTEX()	Creates an instance of a COM object on a remote computer.
N	N	Y	N	N	DEFOLELCID	Specifies the default OLE Locale ID.
N	N	N	N	Y	GETHOST()	Returns an object reference to an Active Document container.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	N	Y	N	N	GETOBJECT()	Activates an Automation object and creates a reference to it.
N	N	N	N	Y	ISHOSTED()	Determines whether an Active Document is hosted in an Active Document container.
N	N	Y	N	N	SET OLEOBJECT	Specifies whether Visual FoxPro searches the OLE Registry when an object cannot be located.
N	N	N	Y	N	SYS(2333)	ActiveX Dual Interface Support.
N	N	N	N	Y	SYS(2334)	Automation Server Invocation Mode.
N	N	N	N	Y	SYS(2335)	Unattended Server Mode.
N	N	Y	N	N	SYS(3004)	Return Locale ID.
N	N	Y	N	N	SYS(3005)	Set Locale ID.
N	N	Y	N	N	SYS(3006)	Set Language and Locale IDs.
N	N	N	N	Y	SYS(4204)	Active Document Debugging.

## Windows/menus

I've lumped all of the commands that have to do with windows and/or menus in one grand list. I haven't marked any of these as obsolete although many are. I've found I hardly ever use these anymore, because the Menu Builder takes care of the menu pieces—and working with forms means that the window functionality usually isn't needed.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	ACTIVATE MENU	Displays and activates a menu bar.
N	Y	N	N	N	ACTIVATE POPUP	Displays and activates a popup.
N	Y	N	N	N	ACTIVATE SCREEN	Sends output to the main VFP window (instead of to the active window).
N	Y	N	N	N	ACTIVATE WINDOW	Displays and activates a window.
N	Y	N	N	N	BAR()	Returns the number of the most recently chosen menu option.
N	Y	N	N	N	CLEAR MENUS	Clears menu bar definitions.
N	Y	N	N	N	CLEAR POPUPS	Clears popup definitions.
N	Y	N	N	N	CLEAR PROMPT	Clears menu items.
N	Y	N	N	N	CLEAR WINDOWS	Clears windows.
N	Y	N	N	N	CNTBAR()	Returns the number of menu options.
N	Y	N	N	N	CNTPAD()	Returns the number of menu pads.
N	Y	N	N	N	CREATE MENU	Creates a menu.
N	Y	N	N	N	DEACTIVATE MENU	Deactivates a menu bar and removes it from the screen.
N	Y	N	N	N	DEACTIVATE POPUP	Deactivates a menu popup.
N	Y	N	N	N	DEACTIVATE WINDOW	Deactivates a window and removes it from the screen.
N	Y	N	N	N	DEFINE BAR	Creates a menu item.
N	Y	N	N	N	DEFINE MENU	Creates a menu bar.
N	Y	N	N	N	DEFINE PAD	Creates a menu pad.
N	Y	N	N	N	DEFINE POPUP	Creates a menu popup.
N	Y	N	N	N	DEFINE WINDOW	Creates a window.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	GETBAR()	Returns the number of an item on a menu.
N	Y	N	N	N	GETPAD()	Returns the menu title for a position on the menu bar.
N	Y	N	N	N	HIDE MENU	Hides a menu bar.
N	Y	N	N	N	HIDE POPUP	Hides a menu popup.
N	Y	N	N	N	HIDE WINDOW	Hides a window.
N	Y	N	N	N	MCOL()	Returns the mouse pointer's column position.
N	Y	N	N	N	MENU()	Returns the name of the active menu bar.
N	Y	N	N	N	MOVE POPUP	Moves a menu to a new location.
N	Y	N	N	N	MOVE WINDOW	Moves a window.
N	Y	N	N	N	MRKBAR()	Determines whether a menu option has a mark.
N	Y	N	N	N	MRKPAD()	Determines whether a menu pad has a mark.
N	Y	N	N	N	MROW()	Returns the mouse pointer's row position.
N	Y	N	N	N	MWINDOW()	Returns the window that the mouse is over.
N	Y	N	N	N	ON BAR	Specifies a menu that is activated when menu option is chosen.
N	N	Y	N	N	ON EXIT BAR	Sets up an event handler for a bar that fires when the user leaves the bar.
N	N	Y	N	N	ON EXIT MENU	Sets up an event handler for a menu that fires when the user leaves the menu.
N	N	Y	N	N	ON EXIT PAD	Sets up an event handler for a pad that fires when the user leaves the pad.
N	N	Y	N	N	ON EXIT POPUP	Sets up an event handler for a popup that fires when the user leaves the popup.
N	Y	N	N	N	ON PAD	Specifies the menu that is activated when a menu pad is chosen.
N	Y	N	N	N	ON SELECTION BAR	Specifies a command that is executed when a specific menu option is chosen.
N	Y	N	N	N	ON SELECTION MENU	Specifies a command that is executed when any menu pad is chosen.
N	Y	N	N	N	ON SELECTION PAD	Specifies a command that is executed when a specific menu pad is chosen.
N	Y	N	N	N	ON SELECTION POPUP	Specifies a command that is executed when any menu option is chosen.
N	Y	N	N	N	PAD()	Returns the menu pad most recently chosen.
N	Y	N	N	N	POP MENU	Restores the specified menu bar.
N	Y	N	N	N	POP POPUP	Restores the specified menu popup.
N	Y	N	N	N	POPUP()	Returns the name of the active menu.
N	Y	N	N	N	PRMBAR()	Returns the text of a menu option.
N	Y	N	N	N	PRMPAD()	Returns the text of a menu pad.
N	Y	N	N	N	PROMPT()	Returns the text for a menu pad or menu option.
N	Y	N	N	N	PUSH MENU	Places a menu definition on a stack.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	PUSH POPUP	Places a menu popup on a stack.
N	Y	N	N	N	RELEASE BAR	Removes a menu bar from memory.
N	Y	N	N	N	RELEASE MENUS	Removes menu bars from memory.
N	Y	N	N	N	RELEASE PAD	Removes a specific menu pad from memory.
N	Y	N	N	N	RELEASE POPUPS	Removes a menu from memory.
N	Y	N	N	N	RELEASE WINDOWS	Removes windows from memory.
N	Y	N	N	N	RESTORE WINDOW	Restores a window.
N	Y	N	N	N	SAVE WINDOWS	Saves window definitions.
N	Y	N	N	N	SET MARK OF	Specifies a mark character for menu titles or menu items, or displays or clears the mark character.
N	Y	N	N	N	SET SKIP OF	Enables or disables a menu, menu bar, menu title, or menu item for user-defined menus or the Microsoft Visual FoxPro system menu.
N	Y	N	N	N	SET SYSMENU	Enables or disables the system menu bar during program execution and allows you to reconfigure it.
N	Y	N	N	N	SET WINDOW OF MEMO	Defines a window for memo editing so it has specific characteristics.
N	Y	N	N	N	SHOW MENU	Displays a menu bar without activating it.
N	Y	N	N	N	SHOW POPUP	Displays a menu without activating it.
N	Y	N	N	N	SHOW WINDOW	Displays windows without activating them.
N	Y	N	N	N	SIZE POPUP	Changes the size of a menu.
N	N	Y	N	N	SIZE WINDOW	Changes the size of a window.
N	Y	N	N	N	SKPBAR()	Determines whether a menu item is enabled.
N	Y	N	N	N	SKPPAD()	Determines whether a menu pad is enabled.
N	Y	N	N	N	SYS(2013)	System Menu Name String.
N	Y	N	N	N	WBORDER()	Determines whether the active or specified window has a border.
N	Y	N	N	N	WCCHILD()	Returns either the number of child windows in a parent window or the names of the child windows in the order in which they are stacked in the parent window.
N	Y	N	N	N	WCOLS()	Returns the number of columns within the active or specified window.
N	Y	N	N	N	WEXIST()	Determines whether the specified user-defined window exists.
N	Y	N	N	N	WFONT()	Returns the name, size, or style of the current font for a window in Visual FoxPro for Windows.
N	Y	N	N	N	WLAST()	Returns the name of the window that was active prior to the current window, or determines whether the specified window was active prior to the current window.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>	<b>Description</b>
N	Y	N	N	N	WLCOL()	Returns the column coordinate of the upper-left corner of the active or specified window.
N	Y	N	N	N	WLROW()	Returns the row coordinate of the upper-left corner of the active or specified window.
N	Y	N	N	N	WMAXIMUM()	Determines whether the active or specified window is maximized.
N	Y	N	N	N	WMINIMUM()	Determines whether the active or specified window is minimized.
N	Y	N	N	N	WONTOP()	Determines whether the active or specified window is in front of all other windows.
N	Y	N	N	N	WOUTPUT()	Determines whether output is being directed to the active or specified window.
N	Y	N	N	N	WPARENT()	Returns the name of the parent window of the active or specified window.
N	Y	N	N	N	WROWS()	Returns the number of rows within the active or specified window.
N	Y	N	N	N	WTITLE()	Returns the title assigned to the active or specified window.
N	Y	N	N	N	WVISIBLE()	Determines if the specified window has been activated and isn't hidden.
N	Y	N	N	N	ZOOM WINDOW	Changes the size and position of a user-defined window or a Visual FoxPro system window.

## Obsolete commands and functions

For the most part, these commands and functions have been superseded by new tools. For example, @SAY and @GET commands were used in FoxBase and FoxPro up to version 2.6 to position objects on a screen. These are no longer needed because of the way that controls are placed on forms in Visual FoxPro. They still work, of course, but they might not in future versions, so don't use them anymore!

In other cases, a command or function is just plain irrelevant. For example, with FoxPro 2.6, you could run the same programs, screens, and reports on a multitude of operating systems. However, because there were differences between those operating systems, you occasionally had to create separate batches of code that would be executed only for a particular platform. For instance, a screen had certain attributes when running under DOS and different attributes when running under Windows. You could determine which operating system was running by using \_DOS, \_MAC, \_WINDOWS and \_UNIX system variables, and then conditionally executing the appropriate code.

Now that Visual FoxPro runs under any operating system as long as “Microsoft Windows” is part of the name, there’s no longer a need to check for a variety of operating systems. These four OS system variables are now obsolete.

This list is provided to forewarn you of commands and functions to stay away from. Hopefully, you’ll be able to avoid the trap of finding some “really cool, undocumented

function" that goes away in the next release, and thus ends up breaking all sorts of your existing code.

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>
Y	Y	N	N	N	@ (MULT)
Y	Y	N	N	N	_ALIGNMENT
Y	Y	N	N	N	_BOX
Y	Y	N	N	N	_CUROBJ
Y	Y	N	N	N	_DOS
Y	Y	N	N	N	_FOXDOC
Y	Y	N	N	N	_FOXGRAPH
Y	Y	N	N	N	_GENGRAPH
Y	Y	N	N	N	_GENPD
Y	Y	N	N	N	_GENSCRN
Y	Y	N	N	N	_INDENT
Y	Y	N	N	N	_LMARGIN
Y	Y	N	N	N	_MAC
Y	Y	N	N	N	_PADVANCE
Y	Y	N	N	N	_PBPAGE
Y	Y	N	N	N	_PCOLNO
Y	Y	N	N	N	_PCOPIES
Y	Y	N	N	N	_PDRIVER
Y	Y	N	N	N	_PDSETUP
Y	Y	N	N	N	_PECODE
Y	Y	N	N	N	_PEJECT
Y	Y	N	N	N	_PEPAGE
Y	Y	N	N	N	_PLENGTH
Y	Y	N	N	N	_PLINENO
Y	Y	N	N	N	_PLOFFSET
Y	Y	N	N	N	_PPITCH
Y	Y	N	N	N	_PQUALITY
Y	Y	N	N	N	_PSCODE
Y	Y	N	N	N	_PSPACING
Y	Y	N	N	N	_PWAIT
Y	Y	N	N	N	_RMARGIN
Y	Y	N	N	N	_TABS
Y	Y	N	N	N	_TRANSPORT
Y	Y	N	N	N	_UNIX
Y	Y	N	N	N	_WINDOWS
Y	Y	N	N	N	_WRAP
Y	Y	N	N	N	ACCEPT()
Y	Y	N	N	Y	ACCESS()
Y	Y	N	N	N	ANSITO OEM()
Y	Y	N	N	N	BARCOUNT()
Y	Y	N	N	N	BARPROMPT()
Y	Y	N	N	N	CALL
Y	Y	N	N	N	CATALOG()
Y	Y	N	N	N	CERROR()
Y	Y	N	N	N	CHANGE()

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>
Y	Y	N	N	N	CLEAR GETS
Y	Y	N	N	N	CLEAR READ
Y	Y	N	N	N	CLEAR SCREEN
Y	Y	N	N	N	CLOSE FORMAT
Y	N	Y	N	N	CLOSE PRINTER
Y	Y	N	N	N	COL()
Y	Y	N	N	N	COMPLETED()
Y	Y	N	N	N	CONVERT
Y	Y	N	N	N	CREATE COLOR SET
Y	Y	N	N	N	CREATE SCREEN
Y	Y	N	N	N	DEFINE BOX
Y	Y	N	N	N	DEXPORT
Y	Y	N	N	N	DGEN()
Y	Y	N	N	N	DISPLAY HISTORY
Y	Y	N	N	N	FILER
Y	Y	N	N	N	FIND
Y	Y	N	N	N	FLDLIST()
Y	Y	N	N	N	INPUT
Y	Y	N	N	N	INSERT
Y	Y	N	N	N	ISCOLOR()
Y	Y	N	N	N	ISMARKED()
Y	Y	N	N	N	JOIN
Y	Y	N	N	N	KEYMATCH()
Y	Y	N	N	N	LKSYS()
Y	Y	N	N	N	LOAD
Y	Y	N	N	N	LOCK()
Y	Y	N	N	N	MDX()
Y	Y	N	N	N	MEMORY()
Y	Y	N	N	N	MENU
Y	Y	N	N	N	MENU TO
Y	Y	N	N	N	NETWORK()
Y	Y	N	N	N	OBJNUM()
Y	Y	N	N	N	OBJVAR()
Y	Y	N	N	N	OEMTOANSI()
Y	Y	N	N	N	ON APLABOUT
Y	Y	N	N	N	ON KEY
Y	Y	N	N	N	ON KEY =
Y	Y	N	N	N	ON MACHELP
Y	Y	N	N	N	ON READERROR
Y	Y	N	N	N	PARAMETERS()
Y	Y	N	N	N	PRINTJOB
Y	Y	N	N	N	PRINTSTATUS()
Y	Y	N	N	N	PROTECT
Y	Y	N	N	N	RDLEVEL()
Y	Y	N	N	N	READ
Y	Y	N	N	N	READ MENU
Y	Y	N	N	N	READKEY()
Y	Y	N	N	N	RELEASE MODULE
Y	Y	N	N	N	RESET (ignored)
Y	Y	N	N	N	RESTORE SCREEN
Y	Y	N	N	N	ROW()

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>
Y	Y	N	N	N	RUN()
Y	Y	N	N	N	RUNSCRIPT
Y	Y	N	N	N	SAVE SCREEN
Y	Y	N	N	N	SCROLL
Y	Y	N	N	N	SET APLABOUT
Y	Y	N	N	N	SET BLINK
Y	Y	N	N	N	SET BORDER
Y	Y	N	N	N	SET BRSTATUS
Y	Y	N	N	N	SET CATALOG ignored
Y	Y	N	N	N	SET CLEAR
Y	Y	N	N	N	SET COLOR
Y	Y	N	N	N	SET COLOR OF
Y	Y	N	N	N	SET COLOR OF SCHEME
Y	Y	N	N	N	SET COLOR SET
Y	Y	N	N	N	SET COLOR TO
Y	Y	N	N	N	SET DBTRAP ignored
Y	Y	N	N	N	SET DELIMITERS
Y	Y	N	N	N	SET DESIGN
Y	Y	N	N	N	SET DEVICE
Y	Y	N	N	N	SET DISPLAY
Y	Y	N	N	N	SET DOHISTORY
Y	Y	N	N	N	SET ECHO
Y	Y	N	N	N	SET FORMAT
Y	Y	N	N	N	SET IBLOCK
Y	Y	N	N	N	SET INSTRUCT
Y	Y	N	N	N	SET INTENSITY
Y	Y	N	N	N	SET LDCHECK
Y	Y	N	N	N	SET MACDESKTOP
Y	Y	N	N	N	SET MACHELP
Y	Y	N	N	N	SET MARGIN
Y	Y	N	N	N	SET MBLOCK
Y	Y	N	N	N	SET MOUSE
Y	Y	N	N	N	SET NOCPTRANS
Y	Y	N	N	N	SET PDSETP
Y	Y	N	N	N	SET PRECISION
Y	Y	N	N	N	SET READBORDER
Y	Y	N	N	N	SET SCOREBOARD
Y	Y	N	N	N	SET SHADOWS
Y	Y	N	N	Y	SET SQL
Y	Y	N	N	N	SET STICKY
Y	Y	N	N	Y	SET TRAP
Y	Y	N	N	N	SET VOLUME
Y	Y	N	N	N	SET XCMDFILE
Y	Y	N	N	N	SHOW GET
Y	Y	N	N	N	SHOW GETS
Y	N	Y	N	N	SHOW OBJECT
Y	Y	N	N	N	SORT
Y	Y	N	N	N	SYS( 3)
Y	Y	N	N	N	SYS( 6)
Y	Y	N	N	N	SYS( 7)
Y	Y	N	N	N	SYS( 12)

<b>Ob</b>	<b>In 2.x</b>	<b>In 3</b>	<b>In 5</b>	<b>In 6</b>	<b>Command/Function</b>
Y	Y	N	N	N	SYS( 13)
Y	Y	N	N	N	SYS( 14)
Y	Y	N	N	N	SYS( 18)
Y	Y	N	N	N	SYS( 21)
Y	Y	N	N	N	SYS( 22)
Y	Y	N	N	N	SYS( 23)
Y	Y	N	N	N	SYS( 24)
Y	Y	N	N	N	SYS( 100)
Y	Y	N	N	N	SYS( 101)
Y	Y	N	N	N	SYS( 102)
Y	Y	N	N	N	SYS( 103)
Y	Y	N	N	N	SYS(1001)
Y	Y	N	N	N	SYS(1016)
Y	Y	N	N	N	SYS(2001)
Y	Y	N	N	N	SYS(2002)
Y	Y	N	N	N	SYS(2003)
Y	Y	N	N	N	SYS(2004)
Y	Y	N	N	N	SYS(2005)
Y	Y	N	N	N	SYS(2008)
Y	Y	N	N	N	SYS(2009)
Y	Y	N	N	N	SYS(2011)
Y	Y	N	N	N	SYS(2016)
Y	Y	N	N	N	SYS(2017)
Y	Y	N	N	N	SYS(2021)
Y	Y	N	N	N	SYS(2024)
Y	Y	N	N	N	SYS(2027)
Y	Y	N	N	N	UPDATE
Y	Y	N	N	N	UPDATED()
Y	Y	N	N	N	UPDATED()
Y	Y	N	N	N	USER()
Y	Y	N	N	N	VARREAD()
Y	Y	N	N	N	WREAD()



# Chapter 3

## The Interactive Data Engine

**The one feature that sets Visual FoxPro apart from the rest of the Visual Studio components is its intrinsic data engine. All of the other tools need to connect to an external data source, and while VFP can also do that, it ships with built-in data access. In this chapter, I'll discuss how relational data works, how VFP physically handles that logical data, and how to use VFP's tools to handle the data structures and the data inside those structures. By the end of this chapter, you'll know how to work with VFP data interactively.**

Data comes in varying sizes and shapes, and one of the primary challenges for programmers since the beginning of computer time has been to store data in an organized fashion. The first widespread attempts, still in use on many mainframes and minicomputers, jammed many types of disparate data into a single file. This methodology, called ISAM, would store a couple of invoices with a header record and details lines like so:

H JONES CONSTRUCTION	JULY 15, 1980	N1200949
D 4 YELLOW WIDGET	\$ 500.00	
D 1 GREEN THINGAMABOB	\$ 12.45	
D 12 RED WHATCHAMACALLIT	\$1,250.00	
H ABC BUILDLING CORP	MAY 4, 1980	N1200950
D 6 BLUE THINGAMABOB	\$ 12.65	

Computer programs, then, had to parse out each line from this file and perform different operations depending on whether the line was a header (lines that began with “H”) or a detail (lines that began with “D”). This was processor-intensive and rather slow. Furthermore, it was very difficult to make changes—suppose the name of the company had to be changed from 35 to 40 characters? Or another element of data had to be included with the header? The rest of the line had to be moved or extended, and the program that handled the file had to be adjusted to accommodate that change—in addition to all of the programs that relied on the old data definition.

As a result, computer scientists labored long and hard to find a better way to handle data. In the early 1970s, two gentlemen named E. F. Codd and C. J. Date proposed a theoretical model for designing data structures; their work—the relational model—has been the foundation upon which new database applications have been created since.

It's important to note that the relational model is a description of the organization of data—not a representation of how the data is stored! Furthermore, you should be aware that the relational model is not a panacea for all data organization. It's excellent for highly structured data with a repetitive format, such as the accounting and sales records of a company. It isn't appropriate for data that has unusual or unstructured characteristics. The contents of all of the CDs of a radio station, a group of multimedia presentations that include sound, video clips, and images, or e-mail (many blocks of widely varying free-form text) are examples that are currently shoehorned, usually with lackadaisical results, into a relational system.

Before you can appreciate the data structures that Visual FoxPro uses in its day-to-day operations, you'll need to understand the relational model of Codd and Date.

## **The relational model**

Related data is contained in a database. This related data may encompass a single application, such as the inventory management system for a retail store, or for a whole company, such as the accounting, sales, production, and human resources data, or anywhere in between.

A database is made up of tables, each of which looks like a spreadsheet—a two-dimensional grid of data elements. Each table stores zero or more instances of an entity—in other words, one or more rows of the spreadsheet. An entity is the generic description of what's being stored in the table, and an instance is a single specific version of the entity.

For example, an inventory management system for a music store might have one table that contains each stock item—CD, tape, or album—that the store carries or can order, and another table that contains each distributor from which the store buys music. The distributor table's entity is “distributor” and each individual distributor is an instance. If there were 200 distributors, there would be 200 instances in the table.

Each column in a table is called a field (or an attribute) and contains a single data element of a single type. This data element must be atomic—that is, it cannot be broken down into smaller pieces, and all of the data elements in a column must be of the same type and contain the same sort of information. For example, columns in the Distributor table might include Company Name, Contact Name, Address, City, State, Zip, Voice Phone, Fax, and Email.

The definition of “atomic” varies according to how the database will be used. The State is an excellent example of an atomic data element—it can't be broken down any further. If you need to store the County within the State, you would use a separate field, but you could still include the State. The Voice Phone might be broken out into three fields—Area Code, Exchange, and Number—but this might not be appropriate if the Distributor table will contain Voice Phones for different countries, because different countries have different structures.

How you define “same sort of information” also can vary. For example, the Email field might contain any kind of e-mail address—Internet, CompuServe, MCIMail, and so on. If it was important, you might need to define different types of e-mail addresses and store them in different fields. However, you wouldn't store Zip Codes, Email Addresses, and Hat Sizes in the same field. (I'll discuss how to handle issues such as these later in this chapter.)

The set of possible values of a column is called the domain of that column. (Imagine my surprise when I found out that the famous “Master of His Domain” episode on the TV sitcom *Seinfeld* wasn't about database programming!) The domain for the State column would be the list of all 50 states in the United States, but could also include the 13 provinces and territories in Canada and the many states in Mexico if the Distributor table was going to include companies throughout North America.

The collection of all the fields in a table make up a record (or a tuple). Each row in a table contains one record and represents a unique instance of that entity. This last point begs repeating. In a properly designed table, each instance in the table—each record—must be unique, by definition. If there are multiple records that are completely identical, then the table was designed incorrectly. There must be a value in at least one of the columns—or a value from a combination of columns—that distinguishes each record from any other record. (If this were not the case, it would be impossible to retrieve that specific record.)

The other important point about an instance is that it can't span more than one record in the table. Often, people would use more than one row in a spreadsheet to store all of the data for a single instance. This is not permitted in the relational model. Use more fields so that the entire instance can be stored in a single record.

Tables are related to each other through the use of attributes (just fields, right?) called keys. Each table must contain at least one column or group of columns that uniquely identifies each instance; if more than one column is used, this groups of columns is called a superkey. (There can be more than one key that uniquely identifies each instance.)

For example, the combination of Name, Address, City, State and Zip Code would make up one superkey; the combination of Phone Number and Zip Code would make up another. There can be more than one superkey in a table, but only one can be defined as the primary key. The choices for a primary key are called candidate keys, and a candidate must contain the very minimum number of columns necessary to make it unique. (Not all candidate keys are superkeys, because a candidate key might be a single column.) For example, if Phone Number and Zip Code make up a candidate key, Phone Number, Zip Code and State do not make up another candidate key, because State is superfluous. If a single column is used to create a key, it is called a single key; if two or more columns are used, the key is called a composite key. The Phone Number/Zip Code column combination would be an example of a composite key, while a Company ID Number would be an example of a primary key made up of a single column.

One of the basic rules of the relational model is the transference of duplicate data out of a table into a second table where it has to exist in only a single record. For example, consider a garden supply store that buys lawnmowers of a certain manufacturer from several different distributors. The Stock table contains a record for each physical lawnmower, and needs to know which distributor a specific lawnmower came from. But you wouldn't want to repeat the company and address information in every Stock record. Instead, all of the Stock items are placed in one table and the Distributors are in another table, and the Distributor's primary key (such as the Distributor ID number) is placed in the Stock table as an attribute. When a primary key for a table appears in another table, it's referred to as a foreign key. Thus, the Distributor ID is called a primary key when it's in the Distributor table and a foreign key when it's in the Stock table.

Obviously, a table can have only one primary key, but it can contain several foreign keys. For example, to expand our garden supply store example, we might have a table that defines a particular brand of lawnmower—model number, price, horsepower, riding or pushing, and whether or not it comes with a refrigerator and a stereo. The Stock table would contain a record for every physical lawnmower; if the store has six identical lawnmowers, the Item table would contain one record whose attributes contained "JX-100", "\$4,300", "1.2 HP", "Riding", "2 cu. Ft refrigerator", and "Bose dual speaker stereo." It would also have a primary key, which might be a randomly generated unique number, like "65094009." The Stock table, on the other hand, would have six records, each of which contained the Item primary key (65094009) as a foreign key, and the Distributor primary key that identifies which Distributor that specific physical lawnmower came from. (It's possible that the garden supply store gets the same model of lawnmower from more than one Distributor, and they want to track where each is coming from. Suppose a particular model had a high defect rate—you'd want to be able to track which Distributor needed to be contacted.)

It's important to remember that the primary key must contain a unique value for each record in the table. This means that it may not be empty or null, and that the method for creating the key for new records must guarantee unique values forever. It's also important to note that while we view records in tables as having "record numbers" much like the row numbers in a spreadsheet, the primary key is the only way a record should be accessed. It is possible to change to "record number" (the relative position) of a record through any number of operations (such as deleting records, physically rearranging the table, and so on), but those operations don't change the value of the keys. In other words, the record number is only a physical convenience and has no meaning in the relational world.

Remember again, the physical representation of these elements—databases, tables, indexes, and so on—don't have to have a one-to-one relationship with their logical cousins. For example, a "database" might be stored in more than one file, or a group of (logical) tables might all be stored in the same physical structure. However, the logical and physical representations often do map to each other. Let's look at how Visual FoxPro physically stores relational data.

## **Visual FoxPro's data structures**

Visual FoxPro traces its heritage to the late 1970s, when an engineer named C. Wayne Ratliff created a program to help him manage football pool statistics at work. The fundamental data structure in his fledgling database management program was a file that held the contents of a single table, but he used an extension of ".DBF", signifying "data base file." As a result, people took to calling each of these ".DBF" files a "database" (instead of the proper handle, "table"), and the misnomer stuck. As a result, two decades of desktop database developers have referred to a single table as a "database" and a collection of tables as "a group of databases."

When Visual FoxPro was introduced, one added feature was a second data structure that contained pointers to .DBF files. This data structure is referred to as a "database container," and it's a more appropriate, if not perfectly accurate, name.

So there we have it—the two fundamental data structures in Visual FoxPro are tables and database containers. There are more, however, so let's examine each of them in turn.

### **Tables**

I'm going to start with tables instead of databases, because tables can exist without a database container. Tables existed before the database container was created, so VFP has to be able to work with them as stand-alone objects.

A Visual FoxPro table can consist of up to three physical files. The table itself has a .DBF extension and contains all of the fixed-length data in the table. An associated memo file has the same name as the table and a .FPT extension, and contains all of the variable-length data in the table. A structural index file has the same name as the table and a .CDX extension, and contains one or more sets of index pointers for the table. A fourth type of file, a stand-alone index file, has an .IDX extension and contains only one set of index pointers.

A table is limited to 2 billion bytes (2 gigabytes), or 1 billion records, whichever limit comes first. In a practical sense, a billion-record table wouldn't be of much interest to people, because the total size is still limited to 2 billion bytes, and so each record would be, er, small. A record can have up to 255 fields and can be up to 64,000 characters in length.

The .DBF file is made up of two pieces: (1) a header record and (2) zero or more data records. The header contains information about the table, such as the number of records in the

---

table, the name of the database it's connected to (if applicable), and flags that indicate whether or not there are associated memo and structural index files. The header also contains a definition of each field in the table, including the name, size, and data type.

Each data record contains a fixed amount of space for each field in the table, plus an additional byte that stores a flag indicating whether or not the record is flagged for deletion, plus one or more bytes to hold information about nulls (which I'll discuss later). As a result, a record with three 10-character fields will actually be 31 bytes (or 32 bytes if nulls are allowed) long. Because the header record takes up a couple hundred bytes as well, you can't calculate the exact size of a .DBF file simply by multiplying the record length by the number of records in it; you have to add the result of the HEADER() function to account for the header size.

Visual FoxPro has two special types of fields—memo and general—whose data is not physically kept in the .DBF file itself. Both of these fields actually contain pointers to locations in the .FPT file for records that have memo or general data, not the data itself. As a result, these two field types can contain very large variable amounts of data—in contrast to the fixed-length fields found in the .DBF file itself. When data is added or modified, it's simply appended to the end of the .FPT file, and the pointer is adjusted as required.

Note that the existence of the memo file or structural index bytes in the table's header record does not prevent you from renaming, moving, or erasing those files from the location on disk. However, doing so will cause error conditions in Visual FoxPro. If you try to access a free table with a missing or corrupted memo file, you will be greeted with a message indicating so, and you will not be able to open the table until the condition is repaired. On the other hand, if you try to access a table with a missing or corrupted structural index file, you will be warned and offered the option to change the structural index byte in the header record. You'll have to rebuild the index, but you can still access the table's data, whereas a bad or missing memo file likely means you'll be missing some of your data. If, however, the table belongs to a database and has a primary key defined, and the index is missing or damaged, you'll get an error and you won't be able open the table at all. In this situation, you've got three choices: (1) Rely on your backup (you did make one, didn't you?), (2) Try to hack the binary structure of the .DBF file yourself, or (3) Pick up a copy of the Stonefield Database Toolkit, which will take care of this and other database-related problems for you.

## Fields

A table can have up to 255 fields. A field can be one of 13 types. The first seven—character, numeric, float, logical, date, memo, and general—were available in the previous version of FoxPro, while the last six—integer, currency, double, datetime, character binary, and memo binary—are new to Visual FoxPro.

Character fields can contain any type of data that can be typed on a keyboard: characters, punctuation, numbers, and so on. The maximum character field width is 254.

Numeric fields can contain numeric data only. Float fields are the same as numeric fields; they were added for dBASE IV compatibility. The maximum width of a numeric or float field is 20 places—and that number includes the decimal point. For example, you could define a numeric field to look like

**1,000,000,000,000,000. (19 places and a decimal)**

or

.0000000000000000000 (a decimal and 19 places)

or

1,000,000,000.000000000 (10 places, a decimal, and 9 more places)

However, this does not mean you'll get 20 places of precision. If you try to fill the above fields completely, you'll get the following results:

```
1,234,567,890,123,400,000      (19 places and a decimal)
0.123456789012346000,0        (a decimal and 19 places)
1,234,567,890.123456000       (10 places, a decimal, and 9 more places)
```

Logical fields contain a logical True or False. You can enter T, F, Y, or N (all case-insensitive). T or F will be displayed if you enter Y or N, respectively.

Date fields contain a date displayed according to user-specified settings but containing a string value formatted as YYYYMMDD. You can manipulate a date by adding and subtracting “numbers of days” to the date to return different dates. You can also extract the various components of the date, such as the month or day of the week, and convert the date value to a text string using native FoxPro functions. Date fields always store the four-digit year even if the display of that date shows only the last two digits, but which century is stored depends on a variety of settings.

Memo fields are variable-length fields that can contain ASCII or binary data. Examples of character data often stored in memo fields include strings that either are longer than the maximum 254 bytes allowed in a character field or whose length varies significantly. Suppose you have to store a long text description in a table, but many records won't have a description at all. If you created a 220-character-wide field, each record would use 220 bytes for that field, whether it contained data or not. Instead of doing so and potentially wasting a lot of space, you could use a memo field to store the description. Each record will require four bytes for the memo field pointer in the table itself, and only records with descriptions will have actual data stored in the memo file.

Examples of binary data stored in memo fields include encrypted data files, binary files such as original copies of programs and reports, and image files. This capability is handy if you want to provide a “copy” of something for the user to manipulate, but don't want them to accidentally (or purposely) modify the original. Storing the original in a memo field and then extracting a copy for the user to work with keeps the original away from harm.

It's important to note that accessing and manipulating data in memo fields isn't as straightforward as using regular fields in a table, but in many cases the flexibility is worth the extra work.

General fields are similar to memo fields in that they can store variable-length data, including binary data. The difference is that the contents of general fields are OLE aware—that is, the data can be understood by another OLE-capable Windows application. For example, a .WAV file created by a sound system recorder would be the data needed to generate a tune, and would be accessed by a sound player to play it.

---

The new fields are generally extensions of existing data types.

Currency fields contain a value that automatically contains 16 integer positions and four decimal positions. The internal representation is different so they only take 8 bytes per record. Double fields allow you to store data that requires a high degree of precision and are most often used in scientific and engineering applications.

Datetime fields contain a combination of the date and time in a single field. The field works like the date field, but to a resolution of seconds. For instance, adding the value 17 to the contents of a datetime field will increase incrementally that field's value by 17 seconds. As with the date field, the entry and display of the date and time in a datetime field can be modified to suit your individual preferences.

The character binary and memo binary fields work similarly to the ordinary character and memo fields except that the data is not translated for different code pages. A code page is a translation table for the high ASCII values according to different languages. For instance, the capital letter "A" is represented by the ASCII code 65. This "65" is stored internally in the computer, and all versions of FoxPro will then display a capital "A" when the "65" code is encountered. However, the codes above ASCII 127 have different meanings for different languages. In English, they are used to display graphic and typographic symbols specific to the English language. In other languages, however, some of those codes between ASCII 128 and ASCII 255 are used for alphabetic symbols specific to other languages—the German umlaut and the French circumflex, for instance.

If you create a table in one code page and then open it in another code page, the character data will be translated automatically. A field containing the ASCII code 204 will display one character for the English code page but a different character for the German code page. Numbers and logicals aren't a problem—they're the same everywhere. But character data is different and will be changed. If you don't want it to be changed, put the data in a binary field instead of the regular field. Examples of fields where you wouldn't want to change code pages are character fields that contain a binary value representing a key or a memo field containing binary information, such as a Word document or an image.

## Nulls

Until this version of FoxPro, handling fields that didn't contain data was somewhat nebulous. FoxPro didn't differentiate between a field that contained no data and a field that contained a zero value. For instance, if you added a blank record but didn't insert data into the fields, a numeric field would be treated as if it had a zero in it. However, this is not technically correct. Because 0 is a value—just like 13 or -219.44—treating a field that didn't contain a value as if the field contained the value 0 was incorrect. Similar circumstances happened with the character and logical data types—logical fields that didn't contain data returned a logical false, and character fields returned an empty string—both of which are different than fields containing no data. Furthermore, testing to determine whether a field had a value was done with the native FoxPro EMPTY() function. However, EMPTY() returned true for numeric fields that contained a 0 and logical fields that contained a logical false.

Visual FoxPro has resolved this situation by providing the ability to designate a field as capable of containing a NULL as well as a value. This designation is part of the field's definition just as its name and type are. NULL is not a data type or a value; rather, it is an

attribute that indicates the presence or absence of data. With NULLs, we have a way to determine if a field has a value or not.

This is often an ethereal concept, so an example might help here. One oft-used analogy is the weather map. The weatherman is making you feel bad by saying it's 78 and sunny in San Diego, 83 and sunny in Orlando, 81 and sunny in Little Rock, but there's no temperature reported in Phoenix. In the database of cities and current temperatures, the value next to Phoenix is not zero, because that would be a valid, if uncomfortable, temperature. Instead, we need to be able to indicate we don't know what the temperature in Phoenix is today—so we use a NULL.

This means when you determine the average temperature, you add 79, 83, and 81, and divide by three, instead of adding 79, 83, 81, and 0, and dividing by four.

When you create a blank record, fields with NULL designations are treated just as fields without NULL designations—character fields are empty, numeric fields contain zero, logical fields contain a logical false, and dates contain an empty date.

However, a field that allows NULLs can have a NULL placed into it. As a result, records with NULLs can be handled differently than records that contain empty strings, zeros, or logical falses. You can think of a NULL as “I don’t know what this value is.” This has ramifications for doing operations on fields that might contain nulls. For example, if you add a value to the contents of a field that contains a NULL, what do you get? You get a NULL, because “something” plus “I don’t know what this value is” is still “I don’t know what this value is.” Sort of like infinity, I suppose, except that maybe it’s smaller. Some operations, like aggregate functions (SUM(), AVG(), and so on) skip records with NULL values.

## **Indexes**

A table contains records in the order they’re added to the table. This order is known as “physical record order” and generally has no meaning in or relationship to the real world. Usually, however, we want to view the records in the table in a specific order—by last name, zip code, date of invoice, or whatever. Instead of changing the physical order of the records, Visual FoxPro uses a second file called an Index file that acts as a mechanism to present the records to us in a different order.

The index file consists of two “fields.” The first field is simply a list of numbers from 1 to the number of records in the table. The second field contains a “pointer” to the record in the table that corresponds to the order in which the records are to be viewed. In the following illustration, the actual records are in order by name, and the date of birth of each individual is in no particular order. The index file enables us to view the records in date-of-birth order without actually changing the physical order of the table.

**Original table**

Table			
Record #			
1	Alexandra	01/01/83	F
2	Bob	12/16/54	M
3	Carl	07/13/27	M
4	Donna	11/01/21	F
5	Ed	11/13/60	M
6	Frank	03/06/70	M
7	George	09/30/34	M
8	Heather	08/05/58	F

**Index file (for order by Birth Date)**

Index #	Table	Record #	
1		4	
2		3	
3		7	
4		2	
5		8	
6		5	
7		6	
8		1	

**Table displayed according to Birth Date index**

Table			
Index #	Record #		
1	4	Donna	11/01/21 F
2	3	Carl	07/13/27 M
3	7	George	09/30/34 M
4	2	Bob	12/16/54 M
5	8	Heather	08/05/58 F
6	5	Ed	11/13/60 M
7	6	Frank	03/06/70 M
8	1	Alexandra	01/01/83 F

The advantages of this separate index file are numerous. For example, it's extremely fast to view records in this order. Instead of physically sorting the file, which would take lots of processing time and possibly a significant amount of disk space, we can use this relatively small index file. Furthermore, changes in the table—for example, a change in a birth date due to a data entry error—can be handled by modifying the pointers in the index table instead of having to physically reorder the entire file. And because the index file is significantly smaller than the table, disk overhead will be smaller. In some cases, the entire index can be cached or brought into memory, making retrieval nearly instantaneous. It's important, however, to note that the index file must be kept up to date as the table data is changed. An index file that isn't in sync with its table will cause more problems than it solves.

More often than not, we'll want to view the table in different orders according to different circumstances. One time we'll want to see the table in date of birth order, another time we'll

want to view the table in order by gender and then alphabetically within each gender. It would seem that we'd need a second index file with a new set of pointers to the record numbers in the table, organized by our new sorting order.

However, there are a number of problems with this approach. First, notice that half the data in each of these separate index files will be identical, because we'll need the Index # (from 1 to 8) each time, and to duplicate the data in each index file would be redundant and wasteful. Second, maintaining separate files means additional work on the part of FoxPro as well as the operating system, which would undoubtedly hurt performance. And additional code in a program rarely makes it more robust.

The solution is to use a single file with a column for Index #, and multiple columns for each set of pointers. In the example above, where indexes are needed for both Birth Date and Sex/Name, our index file would look like this:

### Index file

Index #	Birth Date		Sex/Name	
	Table	Record #	Table	Record #
1		4		1
2		3		4
3		7		8
4		2		2
5		8		3
6		5		5
7		6		6
8		1		7

Setting the index to the second column of pointers would result in a view of the table with all females first, in order by name, and then all males, also in alphabetical order by name.

### Table displayed according to Sex/Name tag

Table	Record #			
	1	Alexandra	01/01/83	F
	4	Donna	11/01/21	F
	8	Heather	08/05/58	F
	2	Bob	12/16/54	M
	3	Carl	07/13/27	M
	5	Ed	11/13/60	M
	6	Frank	03/06/70	M
	7	George	09/30/34	M

Each of these columns that contains a pointer to a table is called a tag, and the expression used to determine the order of the records (in the first case, simply the date of birth, and in the second case, the sex plus the name) is called the index expression.

A single file that contains multiple indexes—or tags—is called a compound index file and has the extension of .CDX. If a compound index file has the same name as the table, it is called a *structural index file* and is automatically opened and maintained when the table is accessed. Nonstructural compound index files must be opened and maintained manually, but still afford the benefits of improved performance and easier maintenance than multiple single index files.

---

You might be wondering why, because a structural index file automatically handles a lot of the details, you'd ever want to use a nonstructural index file. Because an index file is updated on the fly as records are modified, added, and deleted, more tags in an index file will cause more overhead, and might at some point slow performance to an unacceptable level. Instead of suffering through sluggish response time due to an enormous number of tags, it might be desirable to create a series of compound index files for use in different situations.

## **Indexes and Rushmore**

In addition to providing the capability to view tables in various orders, indexes perform another extremely important function—finding specific data. In order to understand how this works, imagine a file cabinet with a separate folder for each company. It would be reasonable to assume that these folders were filed in alphabetical order by company name to facilitate finding a specific company quickly. It would be extremely fast to find the InterGalactic Widgets folder—you'd just skip to the I drawer. However, if the folders were ordered by company name and you wanted to find a company with a specific tax ID number, you'd have to search through every folder until you found it—considerably more time-consuming.

Suppose, however, you knew that you were going to have to look for tax ID numbers as well as company names. You could keep a separate sheet of paper that listed all the tax ID numbers in numeric order, and the company name next to the number. Sort of like our index file, right?

Index files—specifically, tags in index files—are used to rapidly search tables. With some commands, you'll set up the table for searching on a specific tag; with other commands, Visual FoxPro automatically handles the use of tags in each table, as long as the tag exists. The internal mechanism that provides this capability to rapidly search through tables via index tags is called Rushmore, and it's what gives Visual FoxPro its incredible speed and performance.

It's important to mention that tags that can take advantage of Rushmore need to be constructed in a special way. I'll cover how to create Rushmore-optimizable index tags later in this chapter, but the differences between Rushmore searches and non-Rushmore searches are so great that mentioning this several times is justified.

## **Databases**

A database is a table that contains pointers to the data structures in the database—such as tables and indexes. (A database also has other information in it, which I'll bring up when appropriate.) The database table has an extension of .DBC (DataBase Container), the corresponding memo file has the same name as the database but has a .DCT extension, and the structural index has the same name as the database but uses a .DCX extension.

The database table contains records for the database, for each table in the database, for each field in each table, for each index tag, for each persistent relation between the tables (we'll discuss the term "persistent" later), for views, and for connections to external databases. A record for a table contains the path to the table, not simply the table name itself. The entire path, including drive specification, is included if the table is on a drive other than the drive containing the .DBC, but only the relative path is included if the table is on the same drive as the .DBC. Variable-length information about these entities, such as the code associated with a specific database operation, is contained in the database's memo fields.

Because the .DBC is a regular Visual FoxPro table, you cannot access the .DBC table as a regular table with an earlier version of FoxPro. Also, exercise extreme caution when attempting to access the .DBC as a regular table with Visual FoxPro. You can physically access the table, but the contents of the various records and fields have specific meanings and relationships that are maintained by Visual FoxPro when you access the database properly. Inadvertently changing the data might corrupt or invalidate the database and render it unusable.

Once an application's data tables are added to the database, the tables cannot be accessed interactively without opening the database first. When a table is added to a database, a byte in the header record of the table changes to indicate that it now belongs to a database and thus can't be opened without the use of the database. Fortunately, opening a table will also automatically open the database it's part of.

## **.MEM files**

Data can also be stored in .MEM files. These are stand-alone binary files that were used in the past to hold series of memory variables that had to carry over from session to session. Current practice tends toward using a table instead to contain that information. In the past, one tried to minimize the number of tables needed in an application because of the limits to the number of tables that could be open at one time. Now, with restrictions on work areas virtually limitless, it's easier to store, access, and, if necessary, reconstruct those memory variables if a table is used.

## **Other FoxPro files**

A full-featured Visual FoxPro application can contain files with several dozen different file extensions. At first glance, the various extensions seem to be random—or, in some cases, named in order to cause deliberate confusion. While I'm not going to explain in depth what each file does, it's appropriate to list all file extensions and how they relate to each other.

Many components of a Visual FoxPro application use tables as a storage mechanism. In general, you'll find that each component, such as a database, project, report, screen, label, and so on, is contained in a pair of files—a table and a related memo file. The table contains records for each instance of a particular entity, and the memo file contains variable-length information, such as source code or fully qualified file names.

If you're familiar with earlier versions of FoxPro, you'll notice that there is no .SPR or .SPX in the following list. This is not a misprint. Menus still go through the generation process, with the result being an .MPR file that is then compiled into an .MPX, but screens are not handled the same way.

<b>Extension</b>	<b>Description</b>
ACT	Documenting Wizard action diagram
APP	Compiled version of an entire application
BAK	Backup file of a .PRG or .DBF
BAK/TBK	Backup files for a .DBF and .FTP
CHM	HTML Help file
DBC/DCT/DCX	Table, memo, and structural compound index files for a database
DBF/FPT/CDX	Table, memo, and compound index files for a table
DBG	Debugger configuration file
DEP	Setup Wizard dependency file
DLL	Windows dynamic link library file
ERR	Compile error file
EXE	Run-time executable version of an entire application. Visual FoxPro automatically compiles a program (.PRG) file before running it, creating an .EXP. A system can be made up of many .EXPs, each calling another. In order to simplify application distribution, all the .EXPs can be compiled into a single .APP file, which still must be run within Visual FoxPro's interactive environment. An application can also be compiled into an .EXE, which can then be called from a Windows icon, and which relies on the Visual FoxPro run-time files.
FKY	Visual FoxPro macros
FLL	Visual FoxPro dynamic link library
FRX/FRT	Table and memo files for a report
FXP	Compiled version of a .PRG
H	Header (include) file
IDX	Single index file, generally used for backward compatibility
LBX/LBT	Table and memo files for a label
LOG	Coverage log
LST	Documenting Wizard list
MEM	Binary file containing saved memory variables
MNX/MNT	Table and memo files for a menu
MPR/MPX	Generated program and compiled program files for a menu
OCX	ActiveX control
PJX/PJT	Table and memo files for a project
PRG	Text file containing a Visual FoxPro program
QPR	Text file containing a Visual FoxPro query
QPX	Compiled query file
SCX/SCT	Table and memo files for a screen
TMP	Temporary file created by Visual FoxPro
TXT	Generic extension for text file
VCX/VCT	Table and memo files for a visual class library
VUE	View file

## Working with data structures

### Manipulating tables

Tables that are part of a database cannot be accessed without opening the database first. Visual FoxPro will automatically open a table's database if you try to open the table before the database. However, once the database is opened, manipulating one of its tables is pretty much like handling a table that isn't part of a database. Tables that are not part of a database are called *free tables*, and I'll cover how to use them first. Then I'll delve into databases and discuss the additional functionality available for tables that are part of a database.

### Accessing an existing table

Generally speaking, you can access (or “open”) a table simply by entering the command:

```
use MYTABLE
```

in the Command window, where MYTABLE is the name of the table. If you don’t know (or remember) the name of the table, you can enter:

```
use ?
```

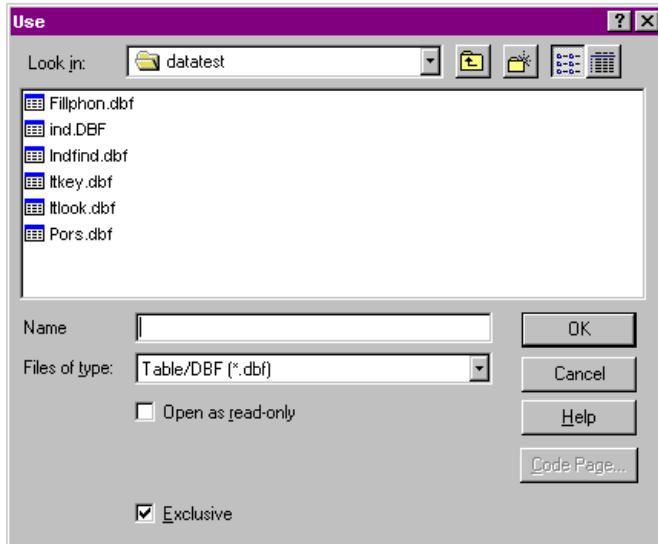
and VFP will present an Open File dialog, through which you can navigate to find the desired table. (See **Figure 3.1.**) If the table is part of a database, the command:

```
use MYTABLE
```

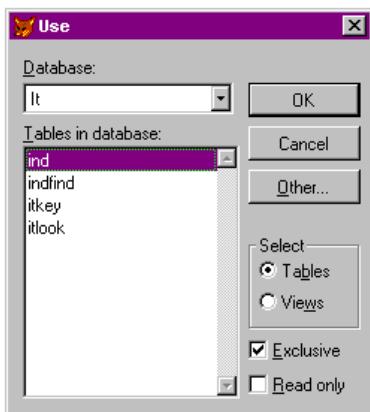
will automatically open the database while opening the table. If a database is open, the command:

```
use ?
```

will display a different dialog—one that shows just the tables contained in the database. (See **Figure 3.2.**) If you want to open a table that’s not in the database, click the Other button and you’ll be presented with the Use dialog in Figure 3.1.



**Figure 3.1.** The USE ? command will present an Open File dialog.



**Figure 3.2.** The USE ? command, when issued when a database is open, will present an Open File dialog that is restricted to the tables in the database.

A number of options are available when opening a table. The Exclusive option gives you exclusive use of the table—no other users can open the table at the same time. The Read-only flag lets you view the contents of the table but not make changes to it. (I'll discuss the difference between Tables and Views later in this book.)

Once you've opened a table for the first time, you might be temporarily disconcerted because the contents aren't automatically displayed. You have to explicitly tell Visual FoxPro to show you the contents. To view data in a table, select the Browse command from the View menu. You'll be presented with a spreadsheet view of the table, with the records appearing in the order that they were physically entered. As with everything in VFP, you can also enter a command. The command:

```
browse
```

will display the same Browse window. I'll examine the Browse window in more detail shortly.

### Exclusive vs. Shared use

Visual FoxPro is natively a multi-user programming environment, which means all the commands and capabilities you need to write multi-user systems are provided in the box. As a result, more than one person on a network can access a table at the same time. This also means a single user can open the same table multiple times on a single machine by running several instances of VFP. I often open several sessions of VFP on a single machine to test multi-user functionality instead of tying up multiple machines on a network.

VFP defaults to opening a table in Exclusive mode, so you'll have to force the table to be opened in Shared mode. You can do this by unchecking the check box in the Open File dialog, or entering this command in the Command window:

```
use MYTABLE shared
```

If you issue the command:

```
set exclusive off
```

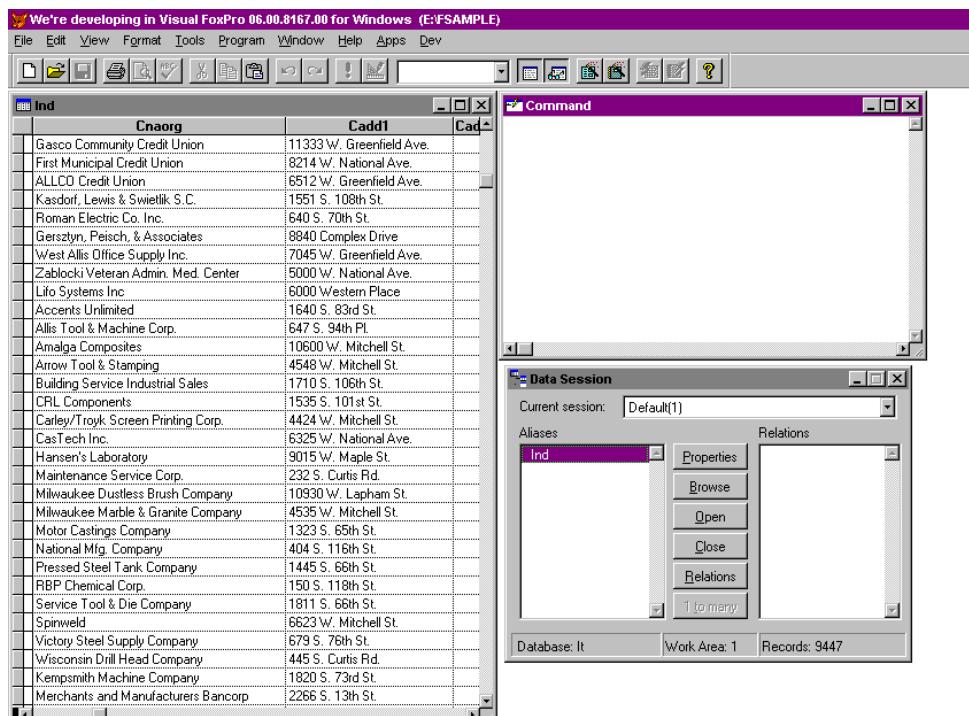
every table for the current session of VFP will be opened shared. (Of course, SET EXCLUSIVE ON will reverse the state.) Note that tables initially opened exclusively are not converted to shared use if you SET EXCLUSIVE OFF afterwards. You have to close the tables and reopen them.

### Accessing multiple tables

Just as you can have multiple spreadsheets or documents open in Excel or Word, you can open more than one table at a time. Unlike Excel or Word, however, you have significant programmatic control over how this is handled. It's easier to follow along if you open up the Data Session window (open the Window menu and click Data Session), or enter this in the Command window:

```
set
```

See **Figure 3.3** for an example of how I usually position the Command and Data Session windows in VFP.



**Figure 3.3.** I usually position the Command window above the Data Session window on the right side of the screen.

---

As you open tables, their names will appear in the left list box of the Data Session window. You can switch between tables, making a specific table “active” by selecting its name in the Data Session window. However, that’s obviously not as useful a mechanism when you’re running a program, and so VFP has the concept of “work areas.”

I’ve often used the analogy of a carnival midway, where somewhere partway down the concrete aisle, there’s the Visual FoxPro booth, manned by a rather dimwitted fellow with thousands and thousands of hands. Every time you tell him to open a table, he grabs a board with the table with one of his many hands. If you want to see the contents of the table, you have to tell him to show you the board (remember, VFP is pretty darn literal). If you tell him to open more than one table, he’ll use hand after hand after hand.

These hands are “work areas,” and you have control over manipulating them. Remember how dimwitted this fellow in the booth is? If he’s got hold of a table (you’ve opened a table and it’s highlighted in the Data Session window), and you tell him to open another table (you issue the USE MYTABLE2 command), he’ll simply drop the one he had open and grab that one instead—so the table you had opened will be closed and the new table will be opened in its place.

In order to open more than one table at a time, you have to select a new work area (tell the guy to use a new hand). There are several variations on how to do this. If you want the new table to be the current table, select an empty work area and then open the table:

```
select 0  
use MYTABLE2
```

Alternately, you could simply tell him to use another hand to open the next table but keep the current table active:

```
use MYTABLE2 in 0
```

Finally, you could also tell him which hand, explicitly, to use, by identifying a work area. Work areas are numbered from 1 to 32,767 (that’s 32K - 1):

```
use MYTABLE in 213
```

But this practice is generally frowned upon because there might already be a table open in that work area—and if so, it would be closed and MYTABLE would be opened in its place.

Once you have tables opened in several areas, you can switch between them, making one or another active. You use the SELECT command to do so (remember that lines beginning with asterisks are comments):

```
* open four tables in unused work areas  
use MYTABLE in 0  
use MYTABLE1 in 0  
use MYTABLE2 in 0  
use MYTABLE3 in 0  
* make MYTABLE2 active  
select MYTABLE2  
* and browse it  
browse  
* make MYTABLE1 active  
select MYTABLE1
```

If you have multiple Browse windows open on the screen, you can move between tables simply by selecting the appropriate Browse window, much as you do with Excel or Word when switching between documents.

You can also give a friendlier name to a table when opening it, and refer to the table by that name. The friendlier name is called an alias:

```
* open four tables in unused work areas
* with friendlier names
use MYTABLE alias ALLY in 0
use MYTABLE1 alias BOBBY in 0
use MYTABLE2 alias INVENTORY in 0
use MYTABLE3 alias EASTSALES in 0
* make MYTABLE2 active
select INVENTORY
* and browse it
browse
* make BOBBY active
select BOBBY
```

This practice of using aliases, rather than specific work area numbers, eliminates a lot of tedious maintenance and makes your code more readable. The previous example is much easier to work with than this:

```
* open four tables in unused work areas
use MYTABLE in 4
use MYTABLE1 in 22
use MYTABLE2 in 16
use MYTABLE3 in 99
* make MYTABLE2 active
select 16
* and browse it
browse
* make MYTABLE1 active
select 22
```

## Closing a table

You can close a table through several different means. If the table is in the current work area, simply issuing the USE command in the Command window will close the table and leave the work area empty. If the table is open in another work area, you can enter the command:

```
use in 7
```

or

```
use in CUSTOMER
```

assuming, of course, that the table is open in work area 7, or that it's named (or given the alias) CUSTOMER.

You can also close a table by opening another one “on top of it”—just open a table while in the work area of the table to be closed. Finally, press the Close button in the Data Session window to close the highlighted table.

## Viewing the contents of a table

Once you have a Browse window open, you can manipulate it much like any other window. In addition, as long as the Browse window is the active window, the Format menu pad disappears while a new pad, Table, appears. (Click on the Command window and you'll see the Table menu pad disappear.)

Naturally, the Browse window can be resized like any other window, and the current record is marked with a small triangle in the leftmost column of the Browse window. The "current record" is an important concept in Visual FoxPro and other Xbase languages and is worthy of some discussion. Unlike other database languages, languages based on the .DBF file structure operate in a "record-centric" manner. (Languages like SQL operate on a "record set" concept—if you want to work with a single record, you need to retrieve a record set consisting of just that one record.) As you manipulate a table, VFP keeps track of the current record by means of an internal "record pointer," much like a word processor keeps track of where you are in the document by positioning a cursor between two characters or under a single character. You can think of the dimwitted fellow in the carnival midway as pointing to a specific row in the table in his hand.

As you move through a table, the value of the record pointer—the number of the current record—changes. When you initially open a table, the record pointer is positioned on the first record, and so the number of the current record is 1. You can see what the current record number is by entering the command:

```
? recno()
```

in the Command window. The number of the current record will be displayed on the screen. Click on a different record in the Browse window, execute this command again, and see the new value. It's important to note that every table has a record pointer and a "current record" even if it's not the active table. I'll cover moving through tables later in this chapter.

The scroll bars on the right side and bottom, if they're visible, allow you to scroll the contents of the Browse window. Note that scrolling the window doesn't actually change the record pointer; you're just changing the "view port" that you're looking at the table from. (Just because you focus your eyes on a different part of the table, the dimwitted fellow hasn't moved his finger.) You can resize columns by dragging the divider bar between the column headings, and move the columns by dragging the column header (click on the column name and then drag). The black box in the lower left corner is a splitter—drag it to the right to provide two panes (or partitions) of the Browse window. The partitions enable you to view two sets of columns in the table that are far apart without having to drag them. You can use the Link Partitions command from the Table menu to synchronize or unsynchronize the left and right partitions.

Pressing the Tab key moves the highlight from field to field in the current record in a left-to-right order (Shift-Tab moves right to left). The left and right cursor keys will move the highlight from field to field when the entire field is highlighted, or from character to character if you are editing the contents of a field. The up and down cursor keys move from record to record, and the PgUp and PgDn keys move the highlight a screen at a time.

By default, a Browse window will display the table in a spreadsheet view; you can view the contents of a single record by selecting the Edit command from the View menu. (You can also enter the EDIT command in the Command window.) As long as an entire field is

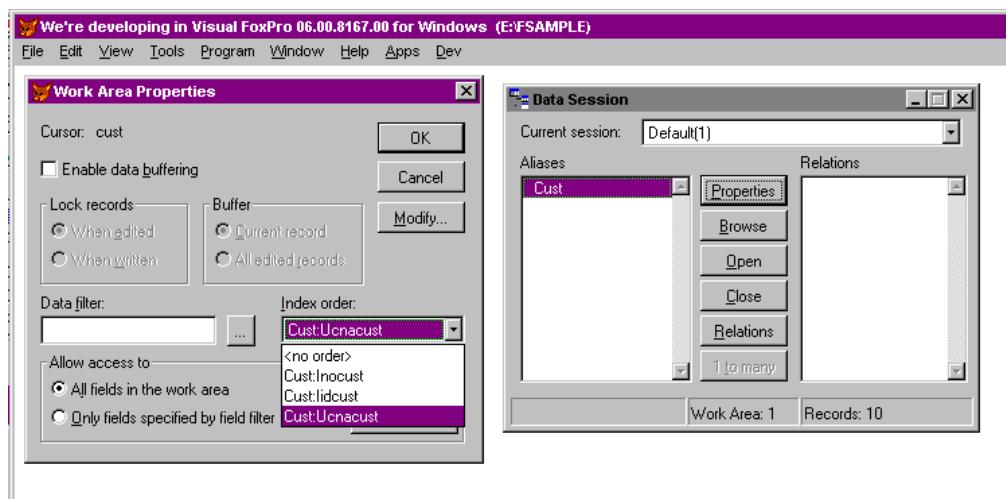
highlighted, the left, right, up and down cursor keys will move the highlight from one field to another, and then, when the end of the record is reached, from one record to the next. If you have a Browse window split into partitions, you can select one of the partitions and change it from Browse to Edit. If your partitions are synchronized, you can scroll through the Browse partition and see the contents of the entire record (or at least the first couple dozen fields, depending on how big your windows are) in the other partition.

### **Specifying the order of records in a table**

Most likely, the records in the table you're working with won't be in the order you want. After all, the records are added to the end of the table. Index tags are used to change the order in which the records are presented to the user. If you know the name of the index tag, you can issue the command:

```
set order to MYTAGNAME
```

Alternatively, if the Data Session window is open, you can click the Properties button to open the Work Area Properties dialog and select the desired tag from the Index Order drop-down. See **Figure 3.4**.



**Figure 3.4.** The Index Order drop-down in the Work Area Properties dialog allows you to change the index order of the selected table.

### **Creating indexes**

The previous section doesn't do you much good if you haven't created indexes, so let's cover that next. Remember that a single index tag is simply a description of the order in which a table will be ordered, and a compound index file (with the .CDX extension) can contain one or more index tags. If a table doesn't already have a .CDX file, it'll be created when you create the first index tag. If it already has a .CDX file, additional index tags will be added to the existing .CDX file.

In order to create an index tag, you must have exclusive use of the table. (If you're not sure if a table is opened exclusively, you can find out in the status bar at the bottom of the VFP window.) You can create an index tag in two ways—interactively or through the Table Designer.

### ***Creating an index tag interactively***

This method requires a comfort level with the commands and functions available in VFP; if you're still getting started, you might want to use one of the next couple of techniques until you have more experience. If you know how to create the index expression, enter the command:

```
index on MYEXPRESSION tag MYTAGNAME
```

where MYEXPRESSION is the index expression and MYTAGNAME is the name of the tag. The index tag name will then appear in the drop-down of the Work Area Properties dialog and you can issue SET ORDER TO MYTAGNAME from then on.

### ***Creating an index tag through the Table Designer***

The Table Designer dialog is used for creating and modifying tables and indexes, and there are about a million ways to get to it. You can enter the CREATE MYTABLE command to create a new table, the MODIFY STRUCTURE command to modify the current table, or click the Modify button from the Work Area Properties dialog. (There are actually two variations of the Table Designer; the other is brought forward when you are creating or modifying tables that belong to a database, and I'll discuss those topics shortly.)

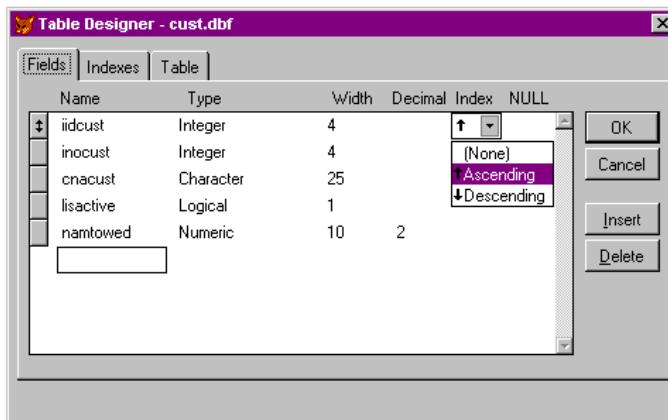
Indexes take one of two forms: simple index tags that just use the field name as the expression—often used for fields that contain a key—and those that use an expression for the index tag. You can create simple index tags using the Fields tab of the Table Designer, but you must use the Indexes tab for tags using more complex expressions.

To create an index on a single field:

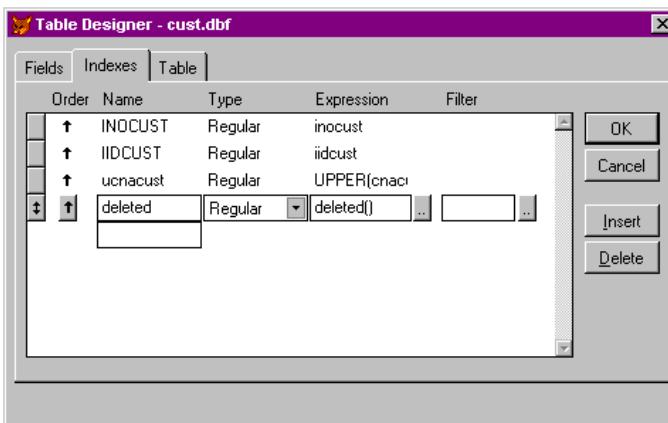
1. Select the Fields tab of the Table Designer. (See **Figure 3.5**.)
2. Highlight the row containing the field upon which you want to create a tag.
3. Click on the drop-down that appears under the Index header, and select either Ascending or Descending. (Or select None to remove the tag.)

To create an index with an expression:

1. Select the Indexes tab in the Table Designer dialog. (See **Figure 3.6**.)
2. Place your cursor in the Name column in the blank row under the last existing index tag. As you start entering an expression, a button under the Order heading will appear with an arrow pointing up. This arrow means that the index will be ascending: A through Z and 0 through 9.



**Figure 3.5.** Build simple tags by using the Fields tab of the Table Designer.



**Figure 3.6.** The Indexes tab of the Table Designer dialog allows you to manipulate indexes for a table. Here, an index on the expression "deleted()" is being created.

3. When you've finished typing the name (it can be a maximum of 10 characters), tab to the next column.
4. Select the Type of index: Regular, Unique, or Candidate. (There is a fourth choice, Primary, for tables that belong to databases.) Generally, you'll want to select Regular. You might find Candidate tags useful for free tables as well as when you want a unique value entered by the user, but that unique value won't be used to relate that table to other tables. For example, you might have a system-generated primary key for a Customer table, but also have a Customer Number field to which you'd assign a Candidate tag.



*You'll want to stay away from Unique index tags. Here's why. A Unique index tag contains one key for all records where the value is the same. For example, if you create a Unique index on "State", you'll have one index tag for "Wisconsin" even if there are two dozen records with the value of "Wisconsin." However, that single index tag is associated with just one of the two dozen records—if you then happen to delete that one record, the index goes away and the other 23 records appear to vanish from the table.*

5. Enter the expression for the index. Alternately, click the gray button to the right of the Expression text box to open the Expression Builder dialog for assistance in constructing a legal expression.
6. Enter the filter expression for the index. Generally, you won't use a filter for index tags built in the Table Designer. Rather, you'll use a filter while programmatically building an index for a special circumstance.
7. Select the order of the index by clicking the arrow button in the Order column. Generally, you'll keep your index tags set to ascending, but you can change a tag to descending (Z through A and 9 through 0) for special circumstances. You can also press the spacebar when the focus is on the button.
8. Create additional tags as desired, and click OK when finished.



*The tag being built in Figure 3.6 might not initially make sense to you. See the discussion about tags on "deleted()" for a full explanation.*

## **Creating a table and modifying the structure of an existing table**

To create a brand new table, type this command in the Command window:

```
create MYTABLE
```

where MYTABLE is the name of the table; or select the New command from the File menu and select the Table option button from the New dialog. In both cases, the Table Designer dialog will appear. This dialog is used to add, delete, and modify fields and index tags for new and existing tables.

For each field, type in a field name. This field name can be a maximum of 10 characters (for tables that don't belong to a database). It must begin with an alphabetic character and can contain only the letters of the alphabet, digits from 0 through 9, and underscores.

Next, select the type of data the field will contain. Depending on your choice of Type, the width and decimal spinners will allow various values. (See the description of fields earlier in this chapter for specific parameters.) Click the NULL button to allow NULLs to be placed in the field.

After you've created at least two fields, the mover bars to the left of the field names allow you to rearrange the order of the fields. By using the Insert and Delete buttons, you can position the cursor in a field and insert a new field in between that field and the one above it, or delete the highlighted field.

Clicking OK will create the table and cause VFP to ask if you want to add records immediately. If you're modifying an existing table, VFP will ask you if you want to save the changes to the structure.

Changing the structure of your table has several ramifications and should be planned carefully. There is no "undo" function to reverse the effects of a structure change. However, the original file is saved with the same name and a .BAK extension. You can rename this file by changing the .BAK extension to .DBF, and then use it as any other table.

Visual FoxPro will let you change the structure of a table regardless of the effects it will have on any data in that table. If fields are deleted, the data in those fields will be lost. If fields are added, empty values will be placed in those fields. If fields are shortened, data will either be truncated (in the case of character fields) or lost (in the case of numeric fields). If fields are lengthened, data will be preserved. If field types are changed, data will be preserved if possible (changing a numeric field to character, for example), or lost if inevitable (changing character to numeric).



*To avoid problems with the conversion of data from one type to another, avoid changing the field type directly. Instead, add a new field, perform a programmatic conversion of the data from the old field to the new one, and then delete the old field. You can have more control over exactly how data is converted than by letting VFP do it for you.*

## **Setting a filter**

You can choose to view a subset of records based on a condition you create, and you can choose to work with a subset of fields as well. For example, suppose that you want to look only at customers located in a specific state. You can create a filter in one of several ways.

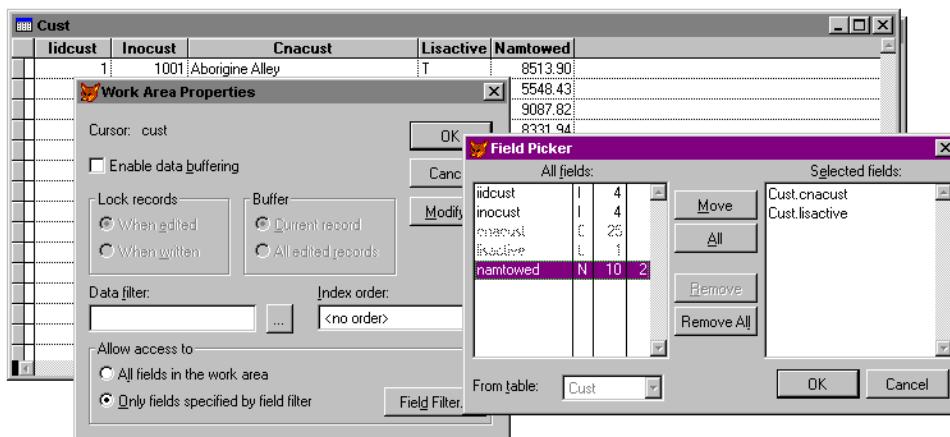
If you know how to create the filter expression, you can enter the command:

```
set filter to MYFILTER
```

where MYFILTER is the filter expression.

Open the Work Area Properties dialog by clicking the Properties button in the Data Session window or by opening a Browse window so that the Table menu pad appears. Open the Table menu and select Properties. In either case, then type an expression in the Data filter text box or click the ellipsis button to bring forward the Expression Builder.

You can also control which fields you want to work with by selecting the "Only fields specified by field filter" option button and then clicking the Field Filter button. (See **Figure 3.7.**)



**Figure 3.7.** Clicking the *Field Filter* button opens the *Field Picker* dialog, which allows you to select which fields you want to work with.

## Manipulating databases

As you learned earlier, tables can be part of a database or they can be independent (“free” tables). Now that you’re comfortable with tables and indexes, I’ll move on to databases, and then cover the differences between tables and indexes that are part of a database.

### Accessing an existing database

There are several ways to access an existing database:

- Open the File menu and select Open. From the Open dialog, select “Database (\*.DBC)” from the Files of Type drop-down. Then navigate through the directory structure to find the desired database.
- Enter the OPEN DATABASE command in the Command window to bring forward the Open dialog (with the Files of Type drop-down already set to “Database (\*.DBC)”) and then navigate through the directory structure as before.
- Enter the OPEN DATABASE MYDB command in the Command window to open the MYDB database.

At the conclusion of any of these steps, the name of the chosen database will appear in the Database drop-down in the standard toolbar. (See **Figure 3.8.**) It’s important to note that opening a database doesn’t automatically open any of the tables in that database.



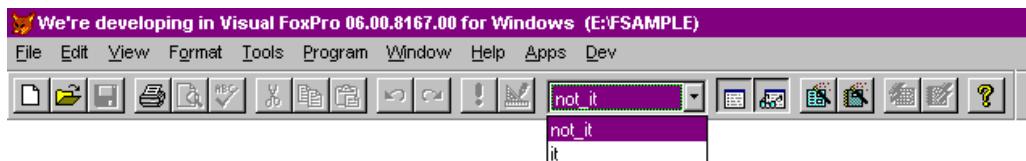
**Figure 3.8.** Opening a database shows the name of the database in the Database drop-down in the standard toolbar. The name of this database is “it.”

### Accessing multiple databases

You can have multiple databases open just as you can have multiple tables open, with the exception that VFP does not use a mechanism like work areas to hold the databases. However, we still have a tool to move between databases, much like we used the Data Session window to visually change from one table to another. The Database drop-down in the standard toolbar will list all open databases, and you can use the drop-down to switch between one database and another. (See **Figure 3.9**.)

You can also use the SET DATABASE TO MYOTHERDB command, where MYOTHERDB is the name of the database you are changing to, in the Command window. After doing so, the name of the database displayed in the standard toolbar will reflect the change.

You can use the DBC() function to determine the current database, and the ADATABASES() function to create a two-column array of all open databases. The first column in the array contains the name of the database and the second column contains the fully qualified pathname.



**Figure 3.9.** You can use the standard toolbar to change from one database to another. Here, the current database is “not\_it,” and I am about to switch to the “it” database.

### Viewing the contents of a database

So, what's in a database? Just as with tables, Visual FoxPro is rather literal-minded when it comes to opening databases: Just because you opened one doesn't mean you're going to see anything special. You have to tell that dimwitted guy behind the counter to show you the contents of the database. (I guess you could think of a database as a big board in the back of the booth upon which all of the tables are going to be hung.) You can use the MODIFY DATABASE command to open the Database Designer. There are several permutations of the MODIFY DATABASE command:

- If you have an open database and you issue MODIFY DATABASE, that database will be displayed in the Database Designer.
- If you issue MODIFY DATABASE IT, the “it” database will be made current (opened if it wasn’t already open) and displayed in the Database Designer. (If another database had been current, it would remain open but not be selected.)
- If you don’t have an open database and you issue MODIFY DATABASE IT, the “it” database will be opened, made current, and displayed in the Database Designer.
- If you don’t have an open database and you issue MODIFY DATABASE, the Open dialog appears, displaying all available databases, just like with the OPEN DATABASE command.

Now that you have the Database Designer open, I should mention that if you explicitly typed the name of the database with the MODIFY DATABASE command, you’ll create a brand new, empty database on disk if it doesn’t exist. So if you’re in the wrong directory or you mistype the name, you’ll end up with a new database file as well—and that can cause needless confusion (you know, as opposed to the confusion that isn’t needless).

Once you’ve successfully issued the MODIFY DATABASE command, three things will happen. First, a window with the name of the database in the title bar will appear. If the database contains tables or views, this window will contain small windows with half-height title bars for each of those tables and views. Second, a new toolbar—the Database Designer toolbar—will appear. If it doesn’t appear, you can bring it forward by opening the View menu and selecting Toolbars, and then checking the Database Designer check box in the Toolbars dialog.

The third thing you’ll notice is that a new Database menu pad appears on the main menu. The menu bars under this menu pad correspond roughly to the buttons on the Database Designer toolbar. There’s also a fourth thing that happens although it’s not readily apparent. Right-clicking inside the Database Designer window opens a context menu that, yes, again, has many of the same options as the main menu and the toolbar. (See **Figure 3.10**.)

Selecting the New Table command will bring forward the Table Designer so that you can define a table. When you’re done, the new table will automatically be added to the database. There are a number of additional capabilities in the Table Designer for tables that belong to a database, and I’ll cover these shortly.

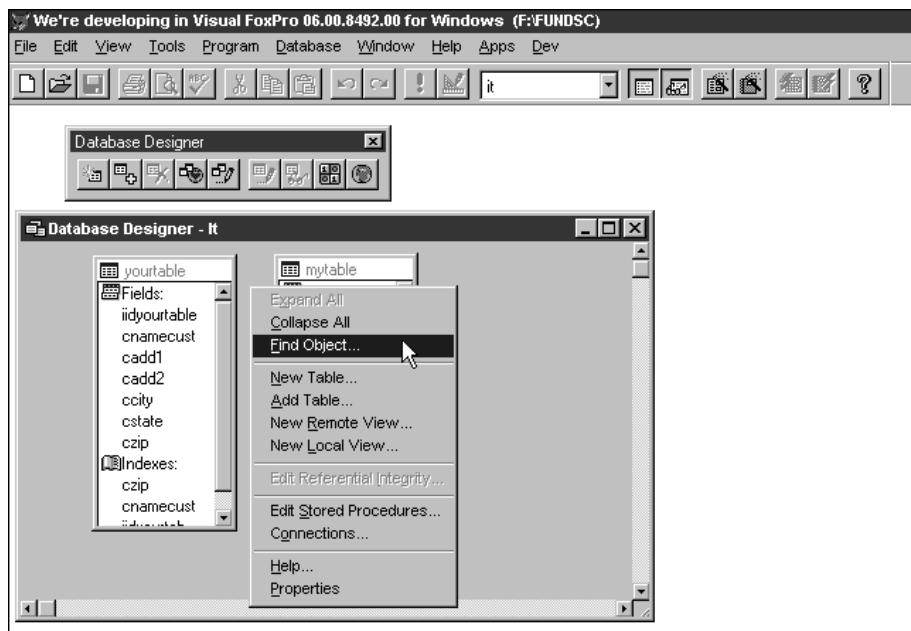
Selecting the Add Table command will add an existing free table to the database; you can’t add a table belonging to a database to another database.

The options having to do with views, stored procedures and connections will be discussed later in this book when they become applicable.

Selecting the Modify Table command brings forward the Table Designer that we’ve already examined. The Remove option removes the table from the database, and, optionally, deletes the file from the disk. There’s no Remove option in the Database Designer context menu—instead, you need to select a specific table in the Database Designer and then right-click to bring up another context menu just for that table—and this context menu contains a Remove option.

The Browse Table command does what you would expect: provides yet another way to open a Browse window for the selected table. This option is available on the Database Designer

toolbar only when a table is selected inside the Database Designer window; it's also available on the context menu of each table in the Database Designer.



**Figure 3.10.** The Database Designer toolbar and the Database Designer context menu have many of the same functions.

### Using the Database Designer

The Database Designer displays all tables in the database, each in its own child window within the Database Designer window. Each table displays a list of all fields in the table, and then below, all indexes in the structural index file for the table. You might also see lines drawn between various table windows—these represent persistent relationships between the tables. You'll notice the relation lines are actually attached to specific indexes in each table. Together, the table window and the relation lines are called the schema of the database and serve to help you visualize the database in its entirety, together with the contents of each table and the relationships between tables.

You can move the table windows around in the Database Designer by dragging the window by its title bar, and resize the window like any other window. Notice that the relation lines follow the table windows to which they belong, much like a leash follows a dog around.

### ***Manipulating a table window inside the Database Designer window***

The table window inside the Database Designer window has several interesting interface capabilities. First, you can browse the table by double-clicking anywhere in the table (yes, I believe that's now the 474th new method for browsing a table). You can resize the table window by dragging any edge or corner. Right-clicking on the table window displays a shortcut menu with the Browse, Delete, Collapse, Modify and Help options. I've covered some of these earlier; the Collapse command reduces the size of the window to display only its title bar. After you do so, the Collapse menu bar in the context menu for that table is replaced by the Expand menu bar.

When you choose Browse or Modify Structure, the table will be opened in the next available work area if it's not already open. If it's open but not in the current work area, VFP will switch to that work area rather than open the table again.

### ***Adding tables to a database***

I've covered the different mechanisms of adding a table to a database, but there are some details you need to be aware of. First, note that you can't add the same table to more than one database—if you try, you'll get a warning message that the table is already part of another database. Why is this? It's because there is a spot in the header of the table for the name of the database it belongs to—so there can be only one database. Selecting the Add Table command displays the Open dialog, from which you can select a table. Note that you don't have a way of visually determining if a table already belongs to a database when you see it in the Open dialog. You also don't know if it's already in the current database unless you can see it in the Database Designer window.

You can navigate through the directory tree to find a table in a different location, but it's advisable to keep all the tables for a database in the same directory. If the table and the database are in the same directory, only the file names are stored in the table header and the database. This means that the database and associated tables are portable—you can move the whole batch of files to another directory and everything works fine. If you have the table and the database in different directories, however, the path of the files (relative if they're on the same drive, absolute if they're on different drives) is stored in the table header and in the database. This severely limits portability.

You don't need to know the name of the database to which a table belongs, because the header record in the table contains the name and path of the .DBC file. However, if you move the tables from one directory to another, those changes are not reflected in the database. Furthermore, if you move the .DBC (and related) files, those changes are not reflected in the table header.

If you've examined the Commands and Functions list in Chapter 2, you probably saw the Add Table command—while you can issue that command in the Command window, you'll probably find it easier to do so through the menu or visually in the Database Designer.

You can use a long file name (more than eight characters) for a table, and you can assign a long name to a table that has an "8.3-style" file name as well. I personally recommend staying away from long names unless you have a compelling reason to use them, but other developers are completely comfortable with them, given a couple of caveats. For example, while you can refer to the table by its long name in your programs, the long name disappears if you remove the table from the database. Any programs that used to refer to that long name will no longer

work. If you use a long name but keep your file names to the 8.3-style format, you can't compare DBF() and ALIAS() successfully. You should, in all cases, avoid spaces in long names because you then have to surround the name with quotes when you open the table, and you will run into some problems.

If you try to add two tables to a database that have the same file name (such as two tables in different directories), you'll be warned that the name is already in the database. (Didn't I already tell you to keep the database and its tables in the same directory? Hmm?) In this case, you must use a long name in order to add it to the database. The long name will be displayed in the half-height title bar of the table window.

### ***Closing a database***

To close a database, issue the CLOSE DATABASE command from the Command window. Doing so also closes all open tables in the database. Note that the Close command in the File menu does not close a database—it just closes the open window. You can verify this by noting that the database name is still visible in the standard toolbar database drop-down, and if you have any tables open from the current database, their names will still be visible in the Data Session window.

The CLOSE DATABASE command works only on the current database and its tables. To close all open databases (and their tables), issue the CLOSE DATABASE ALL command instead. There isn't a way to close a single database if it's not the current database, so don't get frustrated when you're unsuccessful at repeated attempts to CLOSE DATABASE IT.

### ***Creating a database***

You can create a database in one of two ways. The first method is to enter the command CREATE DATABASE in the Command window. If you don't include a file name, you'll be prompted to enter it in the Create dialog that appears.

The second method is to open the File menu and select New, Database, New File. You again will be prompted for a file name in the Create dialog, just as if you'd used the Command window. Once you've entered a name, the database will be created and its name will appear in the database drop-down in the standard toolbar. Unlike creating a table, where you have to define at least one field before you can save it, a database is created as soon as you hit the Enter key. It just won't be a very interesting database because it won't have anything in it.

### ***Manipulating database tables***

I've already discussed how to access a table that's part of a database when the Database Designer window is open. In addition, you can access the table by entering the command USE MYTABLE in the Command window, where MYTABLE is the name of the table. Once the table is open, you can manipulate it—adding, deleting, filtering, and just generally having a great time with it, like you're used to with free tables.

However, there are some caveats and additional capabilities related to manipulating tables that are part of a database. As you're aware, you can't USE a table that belongs to a database without the database automatically being opened. When you enter the USE command in the Command window while a database is open, you'll be presented with a different dialog than that presented when a database is not open, as illustrated in Figure 3.2. It's a good idea to keep

the Data Session window open and the standard toolbar available while you're experimenting with opening databases and tables, to get comfortable with what opens what, when, and where.

### **The Visual FoxPro Data Dictionary**

One of the advantages of a database is that you can attach additional information to the database or to a specific table that enforces how the data in the table is accessed and modified. Because the table can't be accessed except thorough the database, this provides a robust and secure method that also reduces—and in some cases, eliminates—additional coding. For example, you might want to execute a specific operation every time a new record is added to a table. In the pre-Visual FoxPro days, you had to manually call that function each time your program added a record. It was easy to forget to do so, and oftentimes it complicated the program by having multiple copies of the routine in different areas.

Visual FoxPro enables you to perform the following tasks:

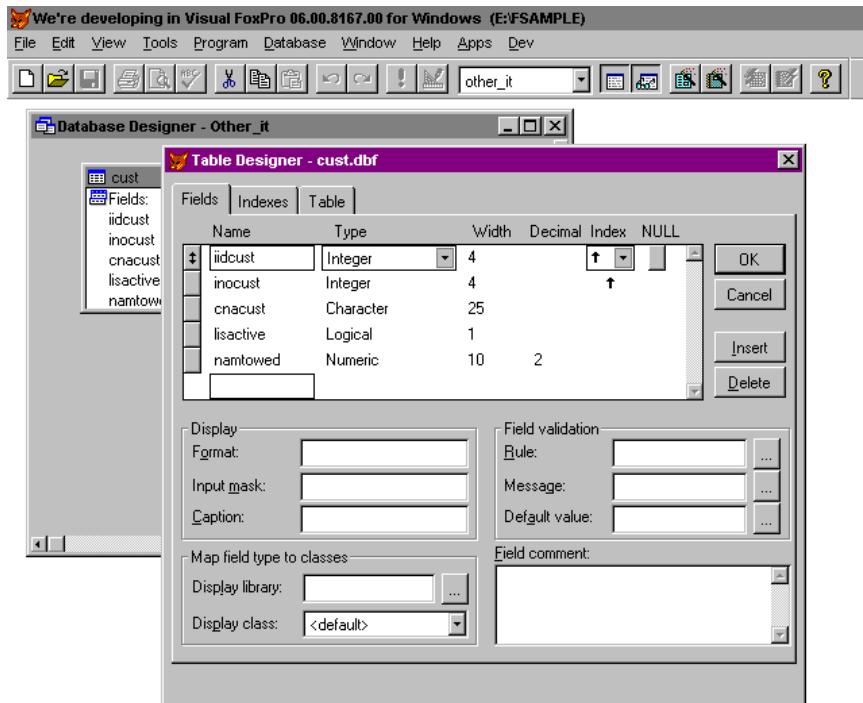
- Create a collection of programs tied to the database.
- Create programs to be automatically executed upon the addition, modification, or deletion of a record.
- Create programs to be automatically executed upon the modification of field values, default values for fields, information related to keys, and relations between tables.

When you issue the MODIFY STRUCTURE command with a table that's part of a database, the Table Designer contains additional objects that it does *not* contain when you modify a free table. (See **Figure 3.11**.)

The first thing that's different is that you can enter field names longer than 10 characters. However, field names longer than 10 characters are truncated if you convert the table to a free table. You might feel that's not a big deal, because once it's in the database, that's that. However, this rule also applies when you pull data into a temporary table, say, through a SQL SELECT command. You can run into some rather unexpected results working with a temp table that has fields with names other than what you thought they were going to be. I've chased down more than one bug because I accidentally named a field name with 11 characters instead of 10, and then got caught when the 11th character wasn't there anymore. My advice: Keep field names to 10 characters.

The second attribute that's different, and, again, not immediately obvious, is the Table tab. There's a Table tab in this dialog for a free table, but it doesn't do anything except show you a few statistics. There's a lot more in the Table tab for tables in a database. (See **Figure 3.12**.)

A record validation is a chunk of code that controls what's allowed to be entered into a record. For example, a rule might be “*price >= cost*.” The record-validation message is a prompt that appears as the error message when a validation rule fails. In the previous example, the text might be “You must enter a price that is at least as much as the cost.” A trigger is a procedure executed upon the specific operation to which the trigger belongs. For example, an INSERT trigger is executed each time a record is added to the table.



**Figure 3.11.** The Table Designer contains additional controls when used with a table that belongs to a database.

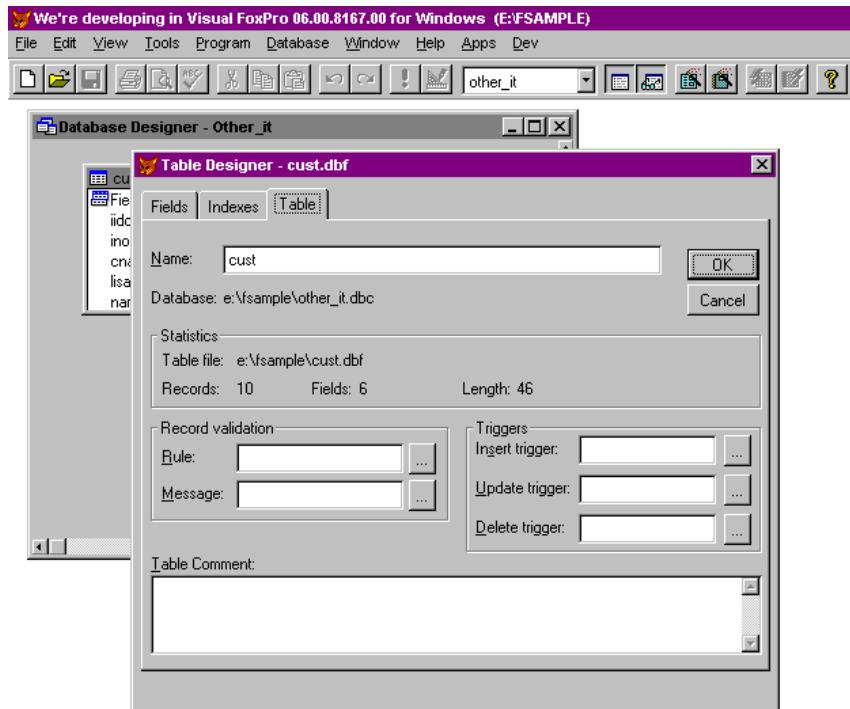
You can also attach validation rules and text to specific fields in the table by using the controls in the lower half of the Fields tab. The validation rule is an expression executed when the user moves off the field, and can be used to ensure that the field contains data satisfying specific criteria. Unfortunately, if the user enters data that doesn't satisfy the validation rule, VFP will display a generic error message that the user might not understand. Fortunately, you can override this behavior because the contents of the Message property will be displayed instead of the VFP generic error message. You can also specify a default value for a field that will automatically be placed into the field when a new record is added to the table. This is particularly handy for generating primary keys and filling audit fields (such as "date record created" and "record created by <user>").

You can also specify default Formats, Input Masks, and Captions. These options are useful because they're used whenever the field is used. The Format specifies the default expression for the display of the field in forms, reports, and Browse windows. The Input Mask specifies the format of the data as it's entered into the field. Captions are used instead of field names in Browse windows, forms, and reports.

The "Map field type to classes" controls will be discussed later in this book when appropriate.

Triggers, validation rules, and default values can also reference programs that are executed—not just expressions. While these programs can reside anywhere that Visual FoxPro

can find them, it's often handy to keep them with the database. In fact, Visual FoxPro (as well as many other true database management systems) has a place to store these programs—and when they are, the programs are called "stored procedures." You can create stored procedures through the appropriate Edit Stored Procedure commands in the main menu, the Database Designer context menu, and the Database Designer toolbar.



**Figure 3.12.** The Table Designer's Table tab has additional controls when used with a table that belongs to a database.

### Working with persistent relationships

You can store relationships between tables in a single database right in the database, similar to stored procedures. These relationships are called "persistent relationships" because they are defined permanently in the database rather than in code. These relationships are used to create default links between tables accessed in the Form Designer and Query Designer. It seems logical that you could create relationships between fields in two tables, because that's where the data is. For example, you might expect to tie the foreign key in one table to its matching primary key in another table. However, persistent relationships are actually tied to indexes. This is because the persistent relationship is used to create the default join condition between the tables, and joins work best on indexed expressions. (Remember when I shook my finger about

indexes and Rushmore?) Furthermore, relations can be based on a complex expression that references more than one field.

### ***Creating persistent relationships***

You can create a persistent relationship between two tables either programmatically or visually within the Database Designer. You cannot use the Data Session window to create persistent relationships, nor can you create persistent relationships between tables in different databases or with a free table.

To create a persistent relationship visually, first ensure that both tables can be seen in the Database Designer window. You might have to adjust the position of one or both to do so. You must also have index expressions in both tables that are identical (although the index tags don't have to be named the same—you can have one tag named "cCustNum" and the other named "cCustomerNumber.") Next, use the mouse to "drag" the index expression in the parent table over the matching index expression in the child table. You won't actually move the index expression, but the mouse action needed for creating the persistent relationship is the same as dragging: clicking, holding the mouse button down, moving to the second index, and releasing the mouse button.

As you do so, you'll see the mouse pointer turn to a horizontal white bar while in the table window, to a circle with a slash through it while moving the mouse pointer over areas that aren't valid index expressions, and then back to the horizontal bar when it is positioned over the new index expression. A line will appear between the two tables, matching the two index names, when you release the mouse button.

To create a persistent relationship programmatically, use the CREATE TABLE or ALTER TABLE commands. These are discussed later in this chapter.

### ***Deleting persistent relationships***

You can also delete persistent relations, either visually programmatically. To do it visually, select the relation line in the Database Designer so it displays in bold, and then press the Delete key. To delete a persistent relationship programmatically, use the ALTER TABLE command, discussed later in this chapter.

## **Working with table structures programmatically**

You can perform all of the above operations—creating tables and modifying their structures, handling persistent relations, and modifying information stored in the data dictionary, such as validation rules—with commands instead of using the interactive interface. This capability is useful both when you can type faster than you can use the mouse, and when you need to perform the operation from within a program. The CREATE TABLE command allows you to create free tables as well as tables that belong to a database; the ALTER TABLE command allows you to change the structure and indexes of free tables as well as the structures, indexes, and data-dictionary information of tables that belong to a database.

### ***Creating tables programmatically***

The CREATE TABLE command allows you to write a command to create a table instead of going through the Table Designer dialog. This is particularly handy when you need to create a table from within a program, such as a temporary table. Some people use an empty table as a

shell for a temporary table, but that method is fraught with danger. Generally, I've found the fewer files you have to ship with your applications, the less maintenance the application will require. If you include an empty table to use as a shell, all sorts of things can happen: you can forget to include it in the next version, accidentally erase it or one of its related files (such as the .FPT), or mechanical problems can corrupt the table. In any of these cases, you're likely to get a support call—one that could be avoided if you simply created the table on the fly.

The CREATE TABLE command is one of the SQL subset commands in Visual FoxPro, and it tends to get rather long. In addition, there's a generally accepted style for writing all SQL commands by continuing them on multiple lines. As a result, we're going to use a possibly unusual style for this and other commands.

The syntax for a basic CREATE TABLE command requires the CREATE TABLE keywords, the name of the table, and a list of the fields (together with the size and type of each field). A typical command would look like this:

```
create table TRANS (iIDTr c(5), cDesc c(20), dBegin d, dEnd d, nAmt n(10,2))
```

A number of additional options allow you to specify a long name (if the table belongs to a database); primary, unique, and foreign key information; and other data. As a result, the command to create a table could get long rather quickly, so we're going to use a special style for these commands. We're going to place each field on a separate line, which will enhance readability and ensure that we don't miss a comma or a parenthesis somewhere. It also makes it easier to add or remove a field in the middle of the command. By the way, as you're practicing with your own CREATE TABLE commands, you might find it easy to type them in the Command window. (Remember, you can use the semicolon to continue lines in the Command window just as you can in programs.) The above command would look like this:

```
create table TRANS ;
(
iIDTr  c(5), ;
cDesc   c(20), ;
dBegin  d, ;
dEnd    d, ;
nAmt    n(10,2) ;
)
```

If you're a C programmer, you'll probably find this style easy to read, and if you're not, just a little practice will make this comfortable as well. You'll see I indent each continued line, which makes reading the program listing easier—you don't have to guess when the next actual line of code starts. Next, note that the field list must be surrounded by parentheses, as is each field width. I've found it far too easy to forget a leading or trailing parenthesis when I don't put the parentheses that surround the entire field list on separate lines, and the Visual FoxPro compiler treats both styles identically.

Next, note that each field is followed by a comma, much as you'd separate items in any list by a comma. However, you don't use any sort of separator between the field name and the type and size. You'll see that commas separate clauses both on the table and field levels, but that commas aren't used within a clause. You'll also notice that the field types and descriptions are lined up, which enhances readability.

Finally, I also put a space between the last character in the line and the semicolon continuation character. Visual FoxPro will “jam” each line together when it compiles the command, and there are some combinations of keywords and clauses that must have a space between them. Instead of trying to memorize which cases require a space and which don’t, I simply ensure that I’ll always have a space where it will be needed.

Now that I’ve covered the basic syntax of the command, it’s time to look at the options. There are two types of clauses available with the CREATE TABLE command. The first type has to do with a specific field and the second type has to do with the table in its entirety. I’ll look at table-level clauses first, and then cover the field-level clauses. Note that many keywords are used in both types of clauses.

### **Table-level clauses**

If you have a database open, creating a table automatically adds the table to the database. If you do not want to include it as part of the database, add the FREE keyword. For example:

```
create table TRANS free ;
( ;
iIDTr  c(5), ;
cDesc  c(20), ;
dBegin d, ;
dEnd   d, ;
nAmt   n(10,2) ;
)
```

The FREE keyword is ignored if you include it when a database isn’t open.

If you have a database open, you can specify a long name for the table, and then use that name to refer to the table. Long names can be up to 128 characters and are required if you have two tables with the same file name as part of the database. For example:

```
create table TRANS name THIS_IS_A VERY_LONG_NAME ;
( ;
iIDTr  c(5), ;
cDesc  c(20), ;
dBegin d, ;
dEnd   d, ;
nAmt   n(10,2) ;
)
```

The NAME keyword has no meaning if a database isn’t open. If you try to include a long name with a table when a database isn’t open, you’ll get a warning message and the command will fail.

You can create primary, candidate, and foreign keys for a table that belongs to a database. Using these clauses with a table that doesn’t belong to a database will generate an error.

You can create a primary key for a table with the PRIMARY KEY clause. You can alternatively create a primary key for a field that would act as the table’s primary key. This clause is used differently when the table’s primary key spans more than one field. A table can have only one primary key, so Visual FoxPro will generate an error if you attempt to use the PRIMARY KEY clause more than once.

For example:

```
create table TRANS ;
(
iIDTr  c(5) primary key, ;
cDesc   c(20), ;
dBegin  d(9), ;
dEnd    d, ;
nAmt    n(10,2) ;
)

create table TRANS ;
(
iIDTr  c(5), ;
cDesc   c(20), ;
dBegin  d(9), ;
dEnd    d, ;
nAmt    n(10,2) ;
primary key str(iIDTr) + cDesc key PKEY1 ;
)
```

You can create a candidate key for the table by using the UNIQUE keyword. Note that the use of the UNIQUE keyword here is not the same as it is when you create a specific UNIQUE index tag. As I mentioned earlier, a table can have multiple candidate keys—and each one must be unique. This is what this UNIQUE keyword is specifying. The UNIQUE keyword with regard to an index creates an index with only one entry for each unique value in the index. Note that you cannot use the same expression/field for a primary key and a unique key.

```
create table TRANS ;
(
iIDTr  c(5), ;
iIDIn  c(5), ;
cDesc   c(20), ;
dBegin  d(9), ;
dEnd    d, ;
nAmt    n(10,2), ;
unique str(iIDTr) + str(iIDIn) tag CKEY1 ;
)
```

You can likewise define a foreign key in the table and establish a relationship to the parent table with the FOREIGN KEY clause. This relationship is defined as persistent and is stored in the database. In this example, a persistent relation is created between the transaction table (the child) and the individual table (the parent) explicitly specifying the key iIDIn. A table can have more than one foreign key, but the expressions must be different. Remember that there must be a comma between each table-level clause.

```
create table TRANS ;
( ;
iIDTr  c(5) , ;
iIDIn  c(5) , ;
cDesc  c(20) , ;
dBegin d(9) , ;
dEnd   d, ;
nAmt   n(10,2) , ;
primary key str(iIDTr) + cDesc tag PKEY , ;
foreign key iIDIn tag FKEY1 reference INDIVIDUAL ;
)
```

### **Field-level clauses**

The field data type is indicated by the letter following the field name. The number inside the parentheses following the field type indicates the width of the column. Some field types don't need a width description because they're always the same. For instance, date fields are always eight characters wide and logicals are always a single character wide. If you specify an invalid field width—say, a width of 13 for a date—Visual FoxPro will ignore it. Numeric, Floating, and Double type fields can also have a decimal width specified. If you don't specify a decimal width, it will default to zero. Remember that the decimal point is part of the entire length, so if you want to be able to store the value 123.45 in a numeric field, you'd use a field width description of 6,2—not 5,2 or 3,2.

You can specify whether a field allows or prevents null values in a field with the NULL and NOT NULL keywords. These keywords must go after the field type and width descriptions, and there is a space between "NOT" and "NULL". If you omit NULL and NOT NULL, the current setting of SET NULL determines if null values are allowed in the field. The NULL and NOT NULL keywords can be used with both free tables and tables that are part of a database.

```
create table TRANS ;
( ;
iIDTr  c(5)  null , ;
cDesc  c(20) not null , ;
dBegin d(9) , ;
dEnd   d, ;
nAmt   n(10,2) ;
)
```

You can specify a validation rule for the field, an error message that Visual FoxPro will display when the field rule generates an error, and a default value for the field. The validation rule can be an expression made up of native Visual FoxPro functions or a user-defined function. Visual FoxPro can't check the validity of a UDF on the fly, so you'll want to be sure to run the command to make sure it works. The error message is displayed when data is changed in a Browse window and it fails the validation rule. It can be a character expression or a UDF. The default value must be of the same type as the field. You must have a CHECK clause if you include an ERROR clause, but you don't need an ERROR clause with a CHECK clause. Note that all these clauses are allowed only in tables that belong to databases.

```
create table TRANS ;
( ;
```

```
iIDTr  c(5), ;
cDesc  c(20) default "BEGINNING BALANCE", ;
dBegin d check ChkBDate(), ;
dEnd   d check ChkEDate(), ;
nAmt   n(10,2) check ChkAmt() error "Amount must be greater than zero" ;
)
```

This capability is important to understand—because Visual FoxPro doesn't have an auto-incrementing field type that you can use to populate a primary key (like Access or SQL Server does), the popular method to create a primary key value is by using a UDF in the Default property for the key field. You can, obviously, add this primary key UDF generator in the Create Table command.

You can prevent the translation of data in a character or memo field to a different code page with the NOCPTRANS keyword. For example, data in the iIDTr field will not be translated if the table is converted to another code page, but the data in cDesc will be.

```
create table TRANS ;
(
iIDTr  c(5) nocptrans, ;
cDesc  c(20), ;
dBegin d(9), ;
dEnd   d, ;
nAmt   n(10,2) ;
)
```

Inclusion of the NOCPTRANS keyword with non-character or memo fields will generate a syntax error.

### Modifying tables and databases programmatically

One of the most often asked questions on the electronic forums regards the need to change the structure of an existing table from within a program. Before Visual FoxPro, the developer had to go through a long, convoluted process that involved creating a new structure, appending the records from the old table, and finally renaming the files as needed. Now, the ALTER TABLE command can be used to modify tables programmatically, much in the same way that CREATE TABLE is used to create tables.

The syntax of the command is straightforward; once you learn a few keywords, you'll likely not refer back to Help again except in arcane situations. You can add, modify, delete and rename columns as well as add and delete attributes of those columns, and add and delete attributes of the entire table.

If you're confused about where in the table each of these clauses is going, execute the command and then bring forward the Table Designer and examine which attributes of the table have changed.

To add and delete a column named lIsAlive, the following commands are used:

```
alter table TRANS ;
add column lIsDead L

alter table TRANS ;
drop column lIsDead
```

To rename the column nAmt and then change its width, the following commands are used:

```
alter table TRANS ;
    rename column nAmt to nFirstAmt

alter table TRANS ;
    alter column nFirstAmt n(8,2)
```

To add and delete a primary key to the table TRANS, the following commands are used:

```
alter table TRANS ;
    add primary key iIDTr key iIDTr

alter table TRANS ;
    drop primary key
```

Note that you don't have to specify the name of the primary key when you drop it because there's only one key per table.

To add and delete a validation rule to the dBegin column of the table TRANS, the following commands are used:

```
alter table TRANS ;
    alter column dBegin set check chkdBegin()

alter table TRANS ;
    alter column dBegin drop check
```

To add and delete a default value for a field, the following commands are used:

```
alter table TRANS ;
    alter column cDesc set default "Beginning Balance"

alter table TRANS ;
    alter column cDesc drop default
```

To add and delete a validation rule to the entire table and provide your own custom error message used when the rule is violated, use the following commands:

```
alter table TRANS ;
    set check chkdTrans() ;
    error "This is your custom error message"

alter table TRANS ;
    drop check
```

Note that the error message will automatically be dropped when you get rid of the validation rule.

Obviously, the ALTER TABLE command is powerful—and dangerous! It is possible to wreak unimaginable havoc in a short period of time through the misapplication of this command. Let's examine what some of the pitfalls are.

First, it's obvious that you should be careful about removing columns that might contain data. What might not be as evident is that if you have index or trigger expressions that reference this field, those rules don't go away. Instead, they'll generate an error at run time.

Second, when renaming columns, you'll want to make sure that you don't have index expressions or rules that reference the original name. Those expressions will not be converted to the new name.

Next, you'll want to be careful when adding validation rules or default values to columns. If the column already contains data, Visual FoxPro will evaluate the validation rule and will issue a warning on every record whose contents violate the rule. Furthermore, the field-validation rules are also executed when you change the width or type of any field.

You can use the NOVALIDATE keyword to specify that Visual FoxPro allow table-structure changes that might violate the integrity of the data in the table. For example, if you specify a .UDF as the default property for a field but that function doesn't exist yet, you'll get an error unless you use this keyword.

## Working with data

That dimwitted fellow in the carnival midway has been patiently waiting for another command to work with the table he's got in his hand, and now is his chance. There are many times that you'll want to move the record pointer to a specific location in a table, and, optionally, find out the value at that location. For example, you might want to determine whether a series of records with a particular value exists, and, if so, update each of those records with a new value, or perhaps display those values for selection by the user. In either case, the important point is that you want to physically move the record pointer in that table to the record in question.

You'll also want to be able to add new records, make changes to existing records, and delete records. I'll cover each of these operations from an interactive point of view and introduce you to the commands that you can use either interactively or programmatically to do these same operations.

## Navigating through tables and finding specific data

You can use the GO and SKIP commands to physically move the record pointer without regard for the data, and you can use the SEEK and LOCATE commands to find a record that has specific data, and to move the record pointer to that record.

### SKIP

SKIP moves the record pointer a specified number of records, either backward or forward through the table. It can also be issued without an argument, in which case the record pointer is advanced a single record. You can use a positive number to advance the record pointer in the table and a negative number to reposition the record pointer backward.

You might be asking yourself, in that devious manner that most developers know all too well, "What happens if I use an argument that's bigger than the number of records in the table?" In general, attempting to skip past the last record or before the first record will position the record pointer at the appropriate end of the file. But there's actually a bit more to it than that, so it's time to talk about the Phantom Record.

### BOF(), EOF() and the Phantom Record

Now that you're comfortable with simple navigation, you need to be aware of the peculiar way that FoxPro structures a .DBF table and the way it handles being at the beginning and end of a file. Before I delve into this conversation, however, I need to introduce a couple of functions

and a new tool—the Watch window—that will make the following discussion much easier to understand.

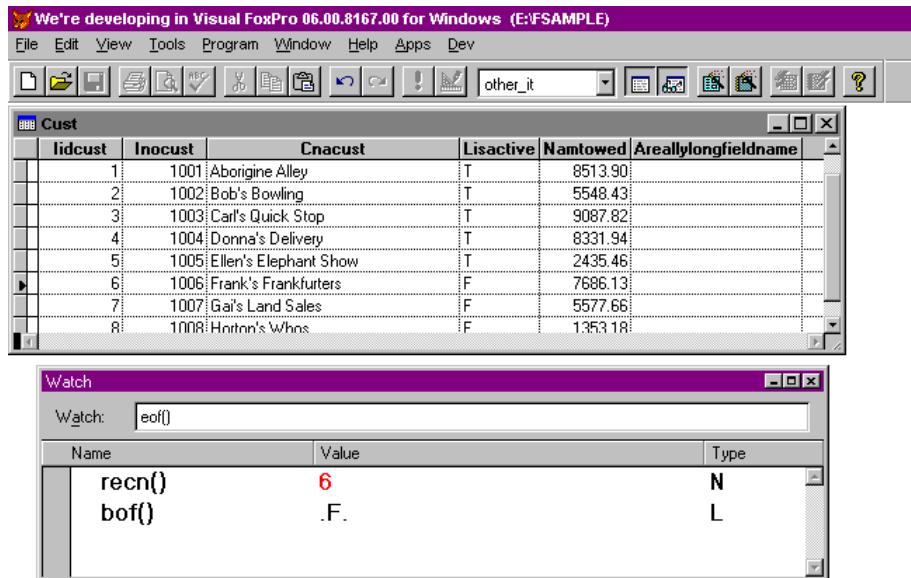
The BOF() and EOF() functions return logical values that indicate whether the record pointer is positioned at the beginning or the end of the file. The RECNO() function, as you already know, returns the current record number.

Visual FoxPro has a sophisticated debugging tool that consists of five windows. I'll discuss the Debugger in much more detail in Chapter 19, but I need to introduce the Watch window now. First, open the Tools menu and select Options. In the Options dialog, select the Debug tab, click the FoxPro Frame option in the Environment drop-down, and then close the dialog. Finally, open the Watch window from the Tools menu. (You can also enter the command ACTIVATE WINDOW WATCH in the Command window.)

You enter expressions in the Watch text box at the top of the Watch window. The expression, together with its evaluated value, will appear in the bottom portion of the Watch window. As you change components that make up the expression, you'll see the evaluated value change in the Watch window. For example, enter the expression:

```
recno()
```

in the Watch window's Watch text box. Then open a Browse window, and as you move the highlight in the Browse window, you'll see the record number (the evaluation of the "recno()" expression) change. See **Figure 3.13**.



**Figure 3.13.** The evaluation of an expression in the Watch window changes as the parts of the expression change. The EOF() function has been typed into the Watch text box but the Enter key hasn't been pressed yet.

For the current exercise, enter the BOF() and EOF() functions in the Watch window if you haven't already. If you don't have a table open, the values 0, .F., and .F. will appear. Open a table with at least four records in it. You'll then see the values 1, .F., and .F. (Remember, when you open a table, the record pointer is automatically placed on the first record.)

If you're paying attention, these results might be disconcerting. Shouldn't the BOF() value evaluate to a true value? After all, we're on the first record!

### ***BOF()***

The BOF() function evaluates to .T. when you try to move the record pointer before the first record in the table. When you opened the table, you were on record number one, and BOF() was .F. Now, try skipping back a record (SKIP -1). Naturally, the record pointer stayed on record number one, but the BOF() condition changed to .T. Essentially, you can think of this function as indicating that you have "bumped up" against the top of the file. When you skip again (forward, that is), you'll move to record number two, and BOF() turns to .F. If you SKIP -1 again, you'll be back on record number one, and BOF() will still be .F. BOF() evaluates to .T. only if you try to move before record number one.

Once you've done something to set BOF() to .T., any further attempts to navigate before record number one (another "SKIP -1", or a "SKIP -500") will generate a "Beginning of file encountered" error. Note that simply being on record number one and doing a SKIP -1 command won't generate the error—if BOF() is .F. You're allowed, while on record number one, to perform a SKIP -1 command once.

### ***EOF()***

Now that I've beat the BOF() function to death, you are probably expecting the discussion for the EOF() function to go rather smoothly. But perhaps there's a slight feeling of dread—after all, what is that "phantom record" mentioned briefly above? Your feeling of dread is well-placed; after all, it's a "phantom" record, right?

Visual FoxPro (and other tools that use a standard .DBF file structure) places a vehicle called the Phantom Record at the end of a file; this record changes the way that EOF() is evaluated. The phantom record acts like an actual record in many situations and allows tables to be related even when one of the tables doesn't have any matching records for a record in the other table. However, it isn't a record, and pretending it like it is will cause early aging and other undesirable side effects.

Suppose your table has 10 records. As you can assume, moving the record pointer to record 10 will return .F. for EOF() because we haven't moved "past" the last record. And, like the events that occur at the beginning of the file, using the SKIP command to move past the last record will flip the EOF() value from .F. to .T. However, the record pointer doesn't stay on record 10—it moves to record number 11, and you can see this in the Watch window. EOF() turns to .T. but RECNO() evaluates to 11—in a 10- record file! This 11th record is the phantom record.

Notice that skipping from record number 10 (when EOF() was .F.) to record number 11 (when EOF() was set to .T.) didn't generate the "End of file encountered" error you might have expected. However, trying to use the SKIP command again (to record number 12, if you want to think about it that way) will do so, because at the moment you are trying to SKIP to 12, EOF() is already .T. This works just like BOF(), doesn't it? The difference, however, is that

while attempting to move before the first record will result in the record pointer staying on record number one, attempting to move past the last record (record number 10) will result in the record pointer moving past the last record and onto the phantom record.

The reason that I've made such a big deal out of this phantom issue is that you have to handle movement at the end of a file differently than at the beginning of the file.

For example, suppose you've provided a mechanism to allow the user to navigate through the table sequentially, using a pair of Next and Previous buttons or arrows. If the user is on the first record, he shouldn't be allowed to press Previous, and if the user is on the last record, he shouldn't be allowed to press Next. You can do this by checking to see if the user is on the first or last record and dimming the appropriate button.

However, you can't just look for "record number one" and "record number NNN" where NNN is the number of records in the table. The user might have changed the order of the records in the table, or set a filter. In either case, "record number one" might not be, currently, the first record in the file. Instead, when attempting to back up through a file, you need to try to move past the beginning of the file and see if the BOF() flag was set to .T.

It isn't quite as easy when we're testing to see if we can move to the next record. Remember, the difference is that when we bump into a true EOF() condition, Visual FoxPro doesn't stay on the same record as it does with BOF(). Instead, the record pointer is moved to the phantom record. Thus, when moving past the end of the file to see if the EOF() flag was set to .T., you'll need to move the record pointer back—off the phantom record and back onto the last real record.

## **GO**

GO moves the record pointer to the record number or position specified by the argument. Arguments can be "TOP", "BOTTOM", or a numeric value. GO TOP moves the record pointer to the first record and GO BOTTOM moves the record pointer to the last record. GO TOP and GO BOTTOM keep BOF() and EOF() at .F., and they both respect the current index order and any filters in effect.

GO N moves the record pointer to record N, if it exists. If it doesn't exist ("N" is larger than the number of records in the table), an error will be generated. You can also simply type a record number in the Command window and press Enter to move the record pointer to that record number. If you enter a record number that doesn't exist, an error will be generated.

## **SEEK**

Here's where the action really starts. In the mid-1990s, it was common to show off FoxPro's prowess as a fast database tool by opening a table that contained every street name in the United States (1.6 million records) on a notebook computer, and then asking someone in the audience for the name of the street they lived on. The speaker would enter a query that asked for every street in the U.S. by that name. A few microseconds later (well, maybe as long as a quarter second, sometimes), a return list of 20, 50, or 200 street names would appear, together with the city and state that the street was in. Yes, that's a successful query of a 1.6-million-record table on a simple notebook computer with a sub-second response time. Old hands at FoxPro would simply yawn, and ask if there was anything new.

SEEK allows you to search for the first record containing a value matching an expression of your choice. In order to use SEEK, you must search for an expression for which there is a

corresponding index tag, and the table must be set to that order before doing the SEEK. (If the table isn't set to any order, using the SEEK command will generate an error.) Suppose you're looking for the street name "WINDWARD" in the cStreetName field in the STREETS table (you'd need an index on cStreetName):

```
m.cSeekString = "WINDWARD"
use STREETS order cStreetName
seek m.cSeekString
```

Now, suppose you weren't sure how the last name was entered into the table. It could have been in proper case ("Windward"), uppercase ("WINDWARD"), or, perhaps in the event that someone had the CAPS LOCK key on by mistake, in reverse proper case ("wINDWARD"). In each of these cases, you'd have to SEEK for the exact manner in which "WINDWARD" was typed into the field. SEEKing on "WINDWARD" when it was entered as "Windward" would not produce a match.

The natural assumption is to convert the entry to the same type each time, such as uppercase:

```
seek upper(m.cSeekString)
```

This wouldn't necessarily work, because you'd still be looking for an uppercase expression in a field that might not contain an uppercase value. The values in the field must match the search expression exactly. In other words, if you wanted to search on UPPER(m.cSeekString), you'd need an index tag on the expression UPPER(cStreetName) in the table. Here is what the code might look like:

```
m.cSeekString = "Windward"
use STREETS
index on upper(cStreetName) tag ucStreetName
set order to ucStreetName
seek upper(m.cSeekString)
```

(Of course, you wouldn't actually index every time you did a SEEK—it was shown just as an illustration.)

Note that the SEEK command finds only the first record that meets the condition. However, because the table is ordered on the expression that you're SEEKing, all of the records that meet the condition will be clustered together. In other words, if you're SEEKing on WINDWARD, all of the records that contain WINDWARD will be grouped together.

## RUSHMORE—revisited

An expression that matches an index tag in a table is referred to as a "Rushmore-optimizable" expression. Searching and querying tables using Rushmore-optimizable expressions is extremely fast, and you should endeavor to do so, in contrast to not using Rushmore-optimizable expressions, whenever possible.

The FOR expressions in the next few commands feature these Rushmore-optimizable expressions.

## LOCATE

The LOCATE command can be used to find any piece of data in one or more records, regardless of the available index tags. If an index tag is available, Visual FoxPro and LOCATE will take advantage of it, but the point of using LOCATE is that you can find any data in a table.

Why not simply create as many index tags as could possibly be needed, and then use SEEK? First, each index tag requires disk space, and complex tags require an inordinate amount of space—space that might be needed more urgently in another part of the application. Second, and more importantly, each index tag imposes overhead because it needs to be maintained every time an add, edit, or delete operation is performed on the file. If a file has dozens and dozens of tags, day-to-day performance can be slowed considerably—and for what benefit? Why maintain tags that are rarely, or even never, used?

The syntax of the LOCATE command requires an expression made up of one or more fields in the table, an operator, and a value. A LOCATE command can have multiple conditions if desired, while a similarly complex SEEK command would need a correspondingly complex index tag. The LOCATE command that corresponds to the earlier search for “WINDWARD” would look like this:

```
m.cSeekString = "Windward"  
locate for cStreetName = m.cSeekString
```

Note that the same situation with possible case problems exists. You could simply convert both the expression and the value to uppercase:

```
m.cSeekString = "Windward"  
locate for upper(cStreetName) = upper(m.cSeekString)
```

As mentioned earlier, one of the benefits to using LOCATE is that you can create multiple conditions. For instance, suppose you wanted to find the Windward street in the northeastern United States (where the zip code began with a “0”). You could use the following command:

```
m.cSeekString = "Windward"  
m.cZipSeekString = "0"  
locate for cZip = m.cZipSeekString ;  
and upper(cStreetName) = upper(m.cSeekString)
```

However, a command with multiple conditions separated by an AND will be evaluated more quickly if the condition most likely to be false is placed first. In other words, suppose you have four conditions all separated by ANDs. If one of those conditions is much more likely to be false, place it at the beginning, because the command will terminate as soon as that condition becomes false. After all, because the definition of an AND is that all of the conditions have to be true, why bother continuing once you find a part that is false?

Note that the LOCATE command finds only the first record that meets the condition. In order to find additional records that meet the condition, use the CONTINUE command repeatedly to find subsequent matches.

## Modifying data

There are three things you generally want to do to the data in a table: add new records, edit existing records, or delete existing records. These three functions together are generally called “maintenance” functions, and Visual FoxPro contains many seemingly redundant commands that provide these capabilities. Some commands are legacies to a smaller and leaner language, while others are newer and more efficient. In addition, some commands are intended to be used in the interactive mode while others are used only inside a program (programmatically). We’ll look at each command and discuss when and where it should be used.

By the way, there are also a series of SQL commands that perform these same operations—I’ll cover them in Chapter 4 along with the rest of VFP’s SQL commands.

### Adding records interactively

You can add a record while in Browse or Edit mode by pressing Ctrl+Y. Doing so will add a record at the bottom of the browse/edit window in which the data for the new record can be entered.

You can use the Append Mode command in the View menu when you have a Browse or Edit window open, to add multiple records without having to press Ctrl+Y for each new record. First, select the Append Mode menu option. A blank record will be added to the table. After entering data into at least one field, pressing the down arrow will automatically add another record. You can continue adding records just by pressing the down arrow as long as you’ve entered data into at least one field of the most recent record. (This requirement exists so those who park a finger on the down arrow key don’t accidentally add 15 blank records by mistake.)

Note that while the record appears to have been added to the bottom of the table, any open index is automatically updated. Once you move the cursor beyond the top or bottom of the Browse window and return to that record, you will see the record in its correct location relative to the other records.

### Adding records programmatically

There are two methods to add records to a table in a program. The first way is to add a blank record to the table, and then replace the empty values in that record with data values from the user. The second way is to use the SQL INSERT command to insert a record already populated with data—as I said earlier, I’ll discuss that in Chapter 4. The first method has been in use since the first version of dBASE II, and is only now giving way to the more efficient insert method.

There really isn’t much to the first method—using the APPEND command to add a record to a table. Period. Open a table, enter APPEND BLANK command. (If you just type APPEND, you’ll be placed in an open editing window on a blank record—which you wouldn’t want to do in a program.) Once the command is finished executing (about a tenth of a blimpsecond later), there’s a new record at the end of the table and it’s the current record. From then on, you can treat it as you would any other record that you edit. See the following section on editing records for information about putting data in an empty record.

### Adding a batch of records from another source

You can also add a group of records from another table or a non-.DBF file with the APPEND FROM command. Note that as of Visual FoxPro 6.0, if DELETED is ON, deleted records aren't added, and if DELTED is OFF, deleted records in the source table are added and the deletion flag stays—in previous versions, the deletion flag was removed once the deleted records were part of the new table.

Suppose you have a table called CUSTOMER that contains all past and present customers for your organization. Another department does a mailing and comes up with a list of potential customers. You don't want to just add that entire list—instead, you'd like to get rid of existing customers first so that they're not added twice.

Your cohort in the other department is an industrious sort and has already gone through the list and deleted the records for current customers. Then you get the list and simply use the APPEND FROM command to add it to your CUSTOMER table:

```
use PROSPECT
* next command returns the value "100"
count
* next command returns the value "77"
* (23 records in PROSPECT where deleted)
count for ! deleted()
select 0
use CUSTOMER
* next command returns the value "500"
count
* next command also returns "500"
count for ! deleted()
* message "100 records added" appears after the following
* command is completed
append from PROSPECT
* returns the value "600"
count
* also returns "600"
count for ! deleted()
* note that the deleted flag for those 23 records
* in the PROSPECT table has been removed when they
* were brought into the CUSTOMER table
```

The COPY TO command can be used to create a new file with a subset of records from the current table. This example copies customers whose last date of purchase was at least 1,000 days ago:

```
use CUSTOMER
* returns the value "500"
count
copy to OLDCUST ;
  for dLastPurch < date() - 1000
sele 0
use OLDCUST
* returns the value "128"
count
```

## **Deleting records interactively**

You can delete a record in Browse or Change mode by pressing the Ctrl+T keystroke combination, or by clicking on the hollow rectangle to the left of the first field in the record.

## **Deleting records programmatically**

You can also delete one or more records by using the DELETE command, including a scope or range of records. Using DELETE without any additional parameters just deletes the current record. The following command would delete the records of all individuals born after Richard Nixon's election:

```
delete for dBIRTH > {11/6/1968}
```

Naturally, deleting records in an actual application is more complex, because we're often going to have to handle business rules along the way. For example, if we delete an organization from the ORG table, how should we deal with the individuals in the IND table related to that organization? Do we let them float free or do we delete them as well? And what about their phone numbers?

Of course, there are related issues when adding records: when we add a person to the IND table, do we have to add a company to the ORG table as well?

These issues all fall under the heading of "Referential Integrity." If you're serious about RI, you'll want to check out the new and very improved RI Builder that Steve Sawyer wrote—see *Effective Techniques for Application Development with Visual FoxPro 6.0* by Jim Booth and Steve Sawyer (Hentzenwerke Publishing, 1998).

Nonetheless, the mechanisms we'll use for adding and deleting don't change—just the environment and rules under which we use those mechanisms.

## **Handling deleted records: recalling and packing**

It's important to know that deleting a record doesn't actually remove it from the table. Rather, it sets a "delete" flag in the record. You can issue the command SET DELETED ON to tell Visual FoxPro to ignore all records marked for deletion when browsing, printing reports, and so on.

You can unmark a deleted record by pressing the Ctrl+T keystroke combination when in a Browse or Edit window, or by clicking on the filled-in rectangle to the left of the first field in a deleted record. You can also programmatically "undelete" by using the RECALL command, which is similar to the DELETE command. For example, the following command will undelete all those records tagged for deletion in the Richard Nixon example:

```
recall for dBIRTH > {11/6/1968}
```

You can permanently remove deleted records from a table by using the PACK command. To PACK a table, you must have Exclusive use of it, either by issuing the SET EXCLUSIVE ON command and then opening the table, or by including the EXCLUSIVE clause to the USE command:

```
use CUSTOMER exclusive  
pack
```

Issuing the PACK command will result in an "Are you sure?" dialog. Note that there's no "unpack" command—packing a table actually performs the following steps:

1. Renames the existing table with a .BAK extension.
2. Copies the structure to a new .DBF file.
3. All records not marked for deletion are appended to the new file.
4. The indexes are rebuilt, and finally the .BAK file is deleted.

It is important to realize that, as a result, there is no “unpack” command. You cannot get the records back—period.



*Many experienced developers don't use the PACK command to remove deleted records because of the potential for data corruption during the process. Instead, they use a routine that essentially does the same thing. This routine looks like this:*

```
use CUSTOMER      && CUSTOMER has some deleted records
copy to TEMPCUSL for NOT deleted()
delete file CUSTOMER.DBF
delete file CUSTOMER.CDX
delete file CUSTOMER.FPT
rename TEMPCUSL.DBF to CUSTOMER.DBF
rename TEMPCUSL.FPT to CUSTOMER.FPT
<run routine to recreate indexes>
```

*(Note that the DELETE command requires the FILE keyword but the RENAME command does not. The ERASE command does the same thing as DELETE but doesn't require the FILE keyword. Isn't Fox lovely—always seven ways to do anything!) The difference is that, as a developer, you control when you delete the original file containing the deleted records and can delete the file once you've made sure that the new table without the deleted records has been created successfully.*



*An alternative to PACKing the table to get rid of deleted records is the use of a technique called "Record Recycling." In this technique, each time a record is deleted, the contents of the record are blanked out. Then, when the user wants to add a new record, the system first looks for an (empty) deleted record. If one is found, the system undeletes it and presents it to the user as a "newly added" record. If the empty deleted record is not found, the system will go ahead and add a brand new record.*

*While this is somewhat more complex, in large systems with a lot of transaction activity, or in systems where gaining exclusive use in order to remove the deleted records is difficult, record recycling can reduce the size of the tables by keeping the number of deleted records to a minimum.*

## Editing records interactively

Editing records interactively is reasonably straightforward. The contents of a Browse or Edit window can be changed simply by typing over them. The contents are written to the table once you have moved to another record (see the following section, “How do I save the data?” for complete details). If you press Escape while still in the field, the original field contents will be preserved.

## Editing records programmatically

Editing a record programmatically involves using the REPLACE or GATHER MEMVAR command to replace the old data with new data.

The REPLACE command has three pieces: the keyword itself, the old and new values, and a scope or range of records upon which the command should operate. For example, to add a new record and stuff values from memory variables into the new record:

```
* create variables that contain data that will be
* placed into the new record
*
m.cNameFirst = "Courtney"
m.cNameLast = "Love"
m.dBirth = {^1967/2/14}
*
* add a new record to the IND table
*
use IND
append blank
*
* now stuff the values from the memory variables into the table
* the "m." indicates a memory variable
*
replace ;
  cNameFirst with m.cNameFirst, ;
  cNameLast with m.cNameLast, ;
  dBirth with m.dBirth
```

The same operation performed with a GATHER MEMVAR command doesn’t save any code, but it relieves typing. REPLACE requires explicit naming of the fields to be updated, while GATHER MEMVAR will stuff the values of any memory values whose names match fields in the table. Here’s the same code using GATHER MEMVAR:

```
* create variables that contain data that will be
* placed into the new record
*
m.cNameFirst = "Courtney"
m.cNameLast = "Love"
m.dBirth = {^1967/2/14}
*
* add a new record to the IND table
*
use IND
append blank
*
* now stuff the values from the memory variables into the table
*
gather memvar
```

Furthermore, typically when GATHER is used, a corresponding command, SCATTER, is used to create a memory variable, or the memory variable is created through a screen. By doing so, adding a new field to a table (and the corresponding screen) doesn't require updating a bunch of REPLACE commands—it comes along for the ride.



*One hidden danger of the GATHER MEMVAR command is that, as it was used in the above example, it doesn't automatically include memo fields. If you have a number of memory variables and one or more are to be gathered into a field's memo fields, you must include the MEMO clause:*

```
gather memo memvar
```

*As a result, I always use "GATHER MEMO MEMVAR" even when the table doesn't have any memo fields—it's safer and the inclusion of the MEMO clause doesn't have any adverse effects when it's unnecessary.*

### **How do I save the data?**

There's a story about a fellow who attended a class in order to learn how to use a spreadsheet, but kept getting called out of the class for this phone call and that meeting. As a result, his knowledge of the tool was less than perfect. About three weeks after the class, the instructor got a panicked call from the fellow because he had run out of room in his spreadsheet. Repeated questioning just turned up answers that made no sense, so the instructor made a courtesy call to the fellow's office.

There, on the computer screen, was the spreadsheet he had been working on—and the bottom row of the display was clearly the middle of some worksheet he had been constructing. The row number at the bottom of the screen was 16384. Quickly paging up, the instructor realized that the fellow had used the same file to build a number of separate spreadsheets—and kept paging down to start each new worksheet.

Evidently the fellow didn't create separate files for each worksheet. After asking why, the instructor's eyes grew wide. This fellow hadn't been in the classroom when the Save command—or the concept of separate files—had been explained, and somehow he had muddled through the rest of the class without picking up the idea. As a result, he had started the spreadsheet program, and just kept adding more data—and had never even saved the file once!

If you're using Visual FoxPro and Visual Studio, you're used to working with Windows, and thus you're envious of a machine that hasn't crashed in three weeks, but try to take your mind off that for a second. The point of this story is that we sometimes take saving data without a second thought. Sure, you know to save your Word documents. But when you get your e-mail, do you execute a "save e-mail" command? Of course not—it just happens automagically.

It's worthwhile to be very explicit when data is saved in Visual FoxPro.

There are two mechanisms for working with tables in Visual FoxPro: direct access and buffering. For the time being, I'm only going to address direct access, or when you have the table opened and you're editing the actual data. I'll address buffering—a more sophisticated method usually used in VFP applications—in Chapter 14, "Your First LAN Application: Building Upon Your Forms."

First of all, when you're in interactive mode, you're using direct access—you're actually editing the data. When you start editing the contents of a field in a record, Visual FoxPro locks that record automatically. No one else can make changes to that record—any field in that record—while you have it locked. However, others can read that record—as it stood before you started editing it—and they can edit other records in the table.

While you are editing the field, you are actually working on a copy of the data in the field. When you leave the field, your changes to the field you are editing are NOT yet written back to the disk. Instead, you need to close the table, leave the record by moving to another record, or execute the FLUSH command to write the contents of the field to disk. If you press Escape while still in the field, your changes will be abandoned and the contents of the field will revert to the original value.

Because this all happens behind the scenes (with the exception of explicitly FLUSHing), it's often not obvious, but it's important to understand this. As long as you're on that record (and haven't FLUSHed it to disk), other users looking at the record might be looking at "yesterday's news"—which obviously has significant ramifications for multi-user database programming.



# Chapter 4

# Visual FoxPro's SQL Implementation

**There are so many parts to Visual FoxPro to love, but if I had to pick one feature to sic a new developer on, it would be SQL. Let's explore.**

SQL is possibly the single most useful feature in Visual FoxPro—even more so than, arguably, the object model. Why? First, Visual FoxPro, more so than other tools in the Visual Studio suite, is data-centric—it's the only full-featured programming language with a native data engine—and SQL's only purpose in life is to “do data.” Second, the skills you learn with SQL are transferable to other tools; you can truly open up a window in SQL Server or Oracle and extract data within minutes once you have a basic understanding of the SQL command set.

There are many things to love about SQL, including speed (properly designed, SQL queries are Rushmore optimized), simplicity, elegance (a single SQL command often can replace dozens of lines of programmatic code), and power (all edit and query interaction with a table can be done with just a few SQL commands). However, a single SQL command can also run several continued lines itself, and the syntax and operation of some clauses can be confusing to use. As a result, the last thing to love about SQL—specifically, its SELECT command—is that you don't have to write out these long commands yourselves: Visual FoxPro has a tool through which you can create many (but not necessarily all) SQL commands through a “point and click” interface. This tool, the RQBE, is an excellent aid to learning much basic SQL syntax.

## SELECT

Many developers spend their whole lives thinking that SELECT is the only SQL command in the FoxPro language—and with good reason. It's undoubtedly the most commonly used command and arguably the most useful. Why? Well, people generally put data into databases because they want to get it out later. While you can use the BROWSE command to view a table or set of tables, generally you will want to pull out only a subset of the data from a database. You can use SELECT to do so.

## Sample databases

Before I get started, I need to describe the tables used as examples in this chapter. In most of this chapter, I'll use a pair of tables, IND and TRANX. IND consists of records for a number of people (individuals), and TRANX contains one or more transactions for those individuals. (I spelled the Transaction table with an “X” because “transaction” is a keyword in Visual FoxPro; using keywords as database, table or field names is generally not a good idea.) IND has the typical fields you would associate with a table of people: first name, last name, address, and so on. TRANX also has the typical data elements for a transaction, including a description, an

amount, beginning date, ending date, and, of course, a foreign key that links the transaction record to a record in the IND table.

I'll use a different pair of tables for the discussion of outer joins, a feature added to Visual FoxPro's SQL implementation in version 5.0. While people and their associated transactions have all the elements needed for most SQL commands, outer joins have a couple of special aspects that are more easily explained through a different data set.

## **Basic SELECT syntax**

A SQL SELECT command consists of about a half dozen clauses that specify which table or tables are being queried, which fields are being pulled from those tables, how to match up tables if more than one is used, which records from those tables are being selected (the record filter), and where to send the result (the destination). As with most things, this will get more involved as we explore the details, but here is the SELECT command in a nutshell.

In its most stripped-down form, a SQL SELECT command that pulls data from a single table consists of two essential clauses. The first clause—using the SELECT keyword—specifies what you are pulling out of the table, and the second clause—using the FROM keyword—specifies what table you are pulling the data from. Visual FoxPro has built-in defaults for the record filter and the destination if they are not provided. An example of the most trivial type of SELECT would be:

```
select * from TRANX
```

which will pull all the fields (denoted by the asterisk) from all the records in the TRANX table and display them in a browse window titled “Query.” If the TRANX table is not opened, it will be opened automatically. It is important to note that SQL commands do not close tables after they finish with them.

## **Field lists**

As we've seen, you can use the asterisk as shorthand to specify all fields in the table. The asterisk will pull all the fields in all the tables if the SELECT is using more than one table. You can also specify to select only certain fields by using a field list in place of the asterisk:

```
select iidTranx, dBegin, dEnd, nAmt from TRANX
```

This will pull only those four fields from the TRANX table and, again, display them in a browse window named Query.

If you are pulling fields from more than one table, you can use just the field name unless that name is present in more than one table. In this case, you need to precede the field name with the table alias. For example, if we were going to pull records from the Transaction table but also wanted the name of the individual to whom that transaction belonged to, we might also want to display the primary key values. And because the Individual table's primary key (iidind) is a foreign key in the TRANX table, it exists in both the Individual and Transaction tables. Thus, it is necessary to specify it, like so:

```
select iidtranx, TRANX.iidind, dBegin, dEnd, nAmt, cNameIn ;
from TRANX, IND ;
where TRANX.iidind = IND.iidind
```

(I'll discuss the WHERE clause in the "Multi-Table SELECTs" section below.)

Forgetting the "TRANX" identifier before the iidind field after the SELECT keyword will result in this error message: "iidind is not unique and must be qualified."

Oftentimes, you will want to pull all the fields from one table but just a couple from another table. Instead of laboriously typing each field name, you can specify all the fields from one of the tables with the table alias and an asterisk, like so:

```
select TRANX.* , cNameIn ;
from TRANX, IND ;
where TRANX.iidind = IND.iidind
```

Even with a consistent naming scheme such as the one used in this book, field names are often terse and cryptic due to the fact that Visual FoxPro only allows them to be 10 characters long for tables that don't belong to a database. Accordingly, the result set of a SELECT may be hard to read with strange-looking column headings. You might wish to use alternate headings with the "AS" keyword:

```
select iidtranx as TRANS_ID, ;
TRANX.iidind as IND_ID, ;
dBegin as BEGIN_DATE, ;
dEnd as END_DATE, ;
nAmt as TRANS_AMT, ;
cNameIn as DEPOSITOR ;
from TRANX, IND ;
where TRANX.iidind = IND.iidind
```

Notice the indenting of this command. It would have been too long to fit on one line, but haphazard line breaks tend to cause more errors than a line that is "neatly" formatted. However, in order to indicate that the second through sixth lines are all part of the same clause, they are all indented an extra space. You'll see this same technique again when I explore multi-line WHERE clauses.

Depending on your personality and the amount of time you have to fine-tune things, you might want to indent the AS part of each phrase so they all line up. If you're having trouble with syntax errors and can't find "that one missing comma," this might be a helpful technique to hunt the errant typo:

```
select ;
iidtranx      as TRANS_ID, ;
TRANX.iidind as IND_ID, ;
dBegin        as BEGIN_DATE, ;
dEnd          as END_DATE, ;
nAmt          as TRANS_AMT, ;
cNameIn       as DEPOSITOR ;
from TRANX, IND ;
where TRANX.iidind = IND.iidind
```

You are not limited to field names in the fields list clause of a SELECT command. You can also use expressions and functions. Suppose you want to determine the number of days between the Begin and End dates of a transaction:

```
select ;
  iidtranx      as TRANS_ID, ;
  dEnd - dBegin as TRAN_DIFF, ;
  nAmt          as TRANS_AMT ;
from TRANX
```

If you want to look up a surcharge based on the amount, you could use a UDF to return the surcharge amount beside the amount:

```
select ;
  iidtranx      as TRANS_ID, ;
  dBegin        as BEGIN_DATE, ;
  dEnd          as END_DATE, ;
  nAmt          as TRANS_AMT, ;
  lupsur(nAmt)  as SURCHARGE ;
from TRANX
```

If you do so, however, you should be aware of two things. First of all, using a UDF will dramatically slow down the SELECT, because no matter how you code it, it's going to be called for every record in the result set. And second, because of this, you'll want to optimize the living daylights out of the function—a savings of a tenth of a second will make a huge difference in a 40,000-record result.

It is important to note that SQL will do a lot of “behind the scenes” work, including opening up temporary tables, indexes, and relations. How and when these operations are done varies with how the SQL optimizer interprets the command. As a result, you cannot assume anything about the environment, such as the current work area, the names of the tables, or even which fields are currently being processed. The only guaranteed method of passing this information to a UDF is to do so through explicit parameters.

A number of SQL SELECT-specific functions (such as COUNT, SUM, and AVERAGE) do aggregate calculations on the result set. I'll discuss these in the “Aggregate functions” section later in this chapter.

## Record subsets

More often than not, you won't want all of the records in the table to land in your result set. To filter out the records you don't want, use the WHERE clause. The WHERE clause is similar to the FOR clause in regular Xbase as far as functionality and basic syntax are concerned. A WHERE clause contains an expression, an operator, and a value. For instance, to select all records with a transaction amount over 100:

```
select ;
  iidtranx      as TRANS_ID, ;
  dBegin        as BEGIN_DATE, ;
  dEnd          as END_DATE, ;
  nAmt          as TRANS_AMT ;
from TRANX ;
where nAmt > 100
```

You can use multiple WHERE clauses by appending them with AND or OR keywords. This command will return all records with an AMOUNT of more than 100 AND whose value in the dBegin field is later than January 1, 1990:

```
select ;
  iidtranx      as TRANS_ID, ;
  dBegin        as BEGIN_DATE, ;
  dEnd          as END_DATE, ;
  nAmt          as TRANS_AMT ;
from TRANX ;
where nAmt > 100 ;
  and dBegin > {1/1/1990}
```

You are not limited to using fields in a WHERE clause that are also in the result set. For instance, you might just want the dates of the larger transactions:

```
select ;
  iidtranx      as TRANS_ID, ;
  dBegin        as BEGIN_DATE, ;
  dEnd          as END_DATE ;
from TRANX ;
where nAmt > 100
```

The WHERE clause is not limited to simple comparison operators. You can also select records where a column has values that fall in or out of a range, that match a pattern of characters, or that are found in a specific list of values. The BETWEEN operator allows you to select a range of values. This operator is inclusive—in this example, records with a Begin Date of 1/1 or 1/31 will be included in the result set:

```
select ;
  iidtranx      as TRANS_ID, ;
  dBegin        as BEGIN_DATE, ;
  dEnd          as END_DATE, ;
  nAmt          as TRANS_AMT ;
from TRANX ;
where between(dBegin, {1/1/90}, {1/31/90})
```

Most programmers are familiar with the asterisk (\*) and question-mark (?) DOS wildcards that stand for “any number of characters” and “just one character.” SQL uses the percent (%) sign to stand for “any number of characters” and the underscore (\_) to represent just one character. One difference, however, is that the DOS asterisk ignores any characters placed after it, while the SQL percent sign does not. In addition, the SQL operator “like” is used with pattern-matching characters. For instance, this will find all records with the word “Visual” somewhere in the field cDesc:

```
select cDesc, dBegin, dEnd, nAmt ;
from TRANX ;
where cDesc like "%Visual"
```

And if the user typed “Visaul” a lot, you could use this command to look for records with any character in the fourth and fifth positions:

```
select cDesc, dBegin, dEnd, nAmt ;
from TRANX ;
where cDesc like "%Vis_1"
```

To test for a string that contains one of these wildcard values, precede the wildcard value with a literal character expression of your choosing, and then use the ESCAPE keyword to identify the literal character expression. This is difficult to imagine, so let’s suppose our Description column in the TRANX table contains a percent sign. Searching for just those records that contain a percent sign somewhere in the Description field with a command like this would meet with a spectacular lack of success:

```
select cDesc, dBegin, dEnd, nAmt ;
from TRANX ;
where cDesc like "%%%"
```

(You’d end up with the entire table in the result set, right?) Instead, define a literal character such as the tilde (~) and then precede the character to search for with the literal:

```
select cDesc, dBegin, dEnd, nAmt ;
from TRANX ;
where cDesc like "%~%" escape "~"
```

The first and third percent signs evaluate to “any number of characters” at the beginning or end of the string, and the tilde plus percent sign character pair evaluates to “the actual percent sign character in the Description field.”

You can also test a column against a list of values. This list might take the form of a hand-typed string of discrete values, or of a result set of a second query. The second type is called a “sub-select” and will be covered in the “Outer joins” section later in this chapter. An example of the first type would be looking for rows where the Description was either a “Beginning,” an “Ending,” or a “Transitory” transaction:

```
select cDesc, dBegin, dEnd, nAmt ;
from TRANX ;
where upper(cDesc) in ("BEGINNING", "ENDING", "TRANSITORY")
```

You can also use the NOT keyword to reverse the effect of an operator, such as BETWEEN, LIKE, or IN. For example, the IND table contains the names and titles of people. If you wanted a list of all individuals who were not Owners, Presidents, or Managers, you could use this command:

```
select cName, cTitle from IND ;
where upper(cTitle) not in ("OWNER", "PRESIDENT", "MANAGER")
```

Like other commands in VFP, string comparisons in SQL are case-sensitive, so a WHERE condition that uses an expression like cTitle = "President" will catch only those records where the title was typed in proper case. All uppercase, lowercase, or mixed case (pRESIDENT) will be ignored. Because it's impossible to train users, and despite my best intentions I never manage to completely convert all data to a consistent state, I've found it safer just to test for the uppercase version of fields to constants also in uppercase.

However, there might be some titles where the word "manager" is not the entire title—and in fact might have several words surrounding it. So you could use a compound WHERE clause, like so:

```
sele cName, cTitle from IND ;
where uppe(cTitle) not in ("OWNER", "PRESIDENT") ;
and upper(cTitle) not like "%MANAGER"
```

If you try this same command with an "OR" instead of an "AND", the entire table will be returned because, when a specific record matches one condition, it will not match the other—thus the entire WHERE clause returns a True value and the row will be included in the result set.

Note that filters you've set with other Visual FoxPro commands, such as SET FILTER TO, are completely ignored by the WHERE clause. However, the setting of SET DELETED is respected. In other words, records tagged for deletion will be ignored if SET DELETED is ON.

The WHERE clause can also be used to join two or more tables together. This is completely discussed in the "Multi-Table SELECTs" section later in this chapter.

## Aggregate functions

You don't have to use SQL to return a collection of records. You might just want to find out "how many" records satisfy a given condition, or find out the highest, lowest, sum, or average value for a column. You can use one of SQL's aggregating functions to do so. The trick to remember here is that even though you are, say, summing values for a group of records, the result set is just a single record that contains the value you were calculating. For instance, to find the sum of the Amount field for all transactions that began in 1994:

```
select sum(nAmt) from TRANS ;
where between(dBegin, {1/1/94}, {12/31/94})
```

This will produce a result set of one record that contains the calculated amount. You can do several aggregate functions at the same time, as long as they are all for the same set of records:

```
select count(nAmt), sum(nAmt), min(nAmt), max(nAmt) from TRANS ;
where between(dBegin, {1/1/94}, {12/31/94})
```

Including fields that don't play into the aggregations will produce nonsensical results:

```
select cIDTr, count(nAmt), sum(nAmt) from TRANS ;
where between(dBegin, {1/1/94}, {12/31/94})
```

This query will place a value in the cIDTr field, but it won't have any meaning to the rest of the result set.

We'll be able to create a "subtotaling" report using these aggregate functions and a descriptive field in combination with the GROUP BY keyword. I'll discuss this next.

## **Subtotaling—GROUP BY**

You've seen that it's possible to create a single record that contains an aggregate calculation for a query—for example, the total of all transactions between two dates. Oftentimes, however, you'd like subtotals as well—say, the total of all transactions for each individual in the system. Use the GROUP BY keyword to do so. The following command will create one record for all transactions in 1994 for each individual ID (cIDIn).

```
select ;
  count(nAmt) ,
  sum(nAmt) ;
from TRANS ;
where between(dBegin, {1/1/94}, {12/31/94}) ;
group by cIDIn
```

Internally, SQL creates a temporary table that contains all fields in the fields list as well as the grouping expression, but just for the records that satisfy the WHERE condition, like so:

```
nAmt, cIDIn
```

Obviously, there might be multiple rows in this temporary table for each individual ID. Then SQL performs the aggregation functions, creating a single row for each unique instance of the individual ID. If the grouping expression isn't one of the fields-list fields (in this case, it isn't), it simply isn't placed in the result set. Typically, however, you'd probably want to include the grouping expression or another identifier in the result set, or else you'd have a series of numbers with no descriptor—technically correct but useless in practice.

## **Multi-Table SELECTs**

As powerful as SQL is, it would be pretty useless if it only operated on a single table at a time. However, multi-table SELECTs really bring out the power and elegance of using SQL in your programs. When writing procedural code, it usually takes about a half-dozen lines of code simply to identify the relationship between two files, and even more code if a third or a fourth file is involved. And, as you know if you've done it, the syntax and order of the commands is somewhat tricky and a nuisance to keep straight. With SQL, however, joining two tables requires only an additional WHERE clause. As you saw in a previous example, the following command will automatically join two tables and pull the name of the individual from the IND table into the result set, along with the data for each selected transaction from the TRANX table:

```
select ;
  iidtranx    as TRANS_ID, ;
  TRANX.iidind as IND_ID, ;
  dBegin      as BEGIN_DATE, ;
  dEnd        as END_DATE, ;
```

```
nAmt      as TRANS_AMT, ;
cNameIn   as DEPOSITOR ;
from TRANX, IND ;
where TRANX.iidind = IND.iidind
```

This special WHERE clause, in which the parent key in one table is equated to the foreign key in the other table, is called a “join condition.” If you name the keys to the various tables with a consistent scheme that makes writing the join condition simple, it’s a painless operation to select data from multiple tables. However, there are three important points to remember about multi-table joins.

First, if you neglect the join condition, SQL will attempt to match every record in one table with every record in the other table; this is known as a Cartesian join. In other words, if TRANX has 100,000 records and IND has 5,000 records, the query will take an extremely long time; the result set, if the query ever finishes, will have 500,000,000 records. This will likely fill up the excess disk space and fray the nerves of the network manager in a hurry.

The second item to remember about joins is how SQL actually matches records. SQL will start from the child side of the relation and find the matching parent. If the parent has no children, the parent record will be left out of the result set. In our example, individuals without transactions will be left out of the result set in the above query. However, if you want all individuals matching a condition, regardless of whether they have transactions, you need to do an outer join, which will be discussed in detail in the “Outer joins” section later in this chapter.

The third warning about multi-table SELECTs concerns joining three or more tables. SQL is not designed to handle a join of one parent and two children. For example, an individual might have several transactions in the TRANX table, and also several phone numbers in the PHONES table. It is not possible to create a single simple SELECT command that will produce a result set that contains individuals with their transactions and their phone numbers at the same time. (You can do it with some advanced techniques, which will be discussed later.) If you try it, you will end up with a result set from the parent and one of the children—whichever has more records. The query might seem to work, but the answer created will be wrong.

## Outer joins

As I mentioned a couple paragraphs earlier, an ordinary SQL join will match only records that have both a parent and a child. These are called “inner joins.” If you want SQL to include records on one side or another that don’t have a corresponding record on the other side, you need to create an “outer join.”

In versions prior to VFP 5.0, you had to resort to a bit of trickery to make an outer join happen. Most SQL implementations use a NULL record to act as a “placeholder” for the records with missing children. With earlier versions of Fox, however, people actually created two queries—one that collects all the parents with children, and a second that collects all the parents without children—and then combined those two collections into a final result set.

With VFP 5.0 and later, true outer join support was available, but I’ll discuss the two-query technique because you might run across existing applications that use it, and it can be useful elsewhere. The tables I’ll use in the next few examples are shown in **Figure 4.1**.

**Ind**

iidind	Cnaf	Cnal
1	Al	Aggressive
2	Bob	Belligerent
3	Carla	Courteous

**Tranx**

lidtranx	iidind	Cdesc	Namt	Dbegin	Dend
1	1	Groceries	100.00	09/18/1998	09/23/1998
2	2	Widgets	200.00	06/10/1998	06/20/1998
3	2	Movie Tickets	34.00	03/02/1998	03/17/1998
4	2	New Muffler	250.00	11/22/1997	12/12/1997
5	0	Orphan Transaction	99.95	04/02/1998	04/03/1998

**Figure 4.1.** The IND and TRANX tables.

The following query—yes, it's a long one—is the old way of combining two tables when you wanted to include all records regardless of if they had a child for every parent:

```

select ;
  cNaF      as DEPOSITORF, ;
  cNaL      as DEPOSITORL, ;
  dBegin    as BEGIN_DATE, ;
  dEnd      as END_DATE, ;
  nAmt      as TRANS_AMT ;
from TRANX, IND ;
where TRANX.iidind = IND.iidind ;
union all ;
select ;
  cNaF      as DEPOSITORF, ;
  cNaL      as DEPOSITORL, ;
  {}        as BEGIN_DATE, ;
  {}        as END_DATE, ;
  00000.00  as TRANS_AMT ;
from IND ;
where iidind not in ;
  (select distinct iidind from TRANX)

```

The first query (everything up to the UNION clause) isn't too bad—it just pulls all individuals with transactions. The second query, however, is much more complex and actually does two things. The subselect query in the WHERE clause puts together a list of all the individuals with transactions—by finding their foreign keys. Then the main SELECT command pulls all individuals where the individual ID is not in the subquery list—in other words, those individuals without transactions.

The UNION keyword requires that the structures of the two result sets must be identical. Because the IND table does not have dBegin, dEnd, or nAmt fields (in the second query), we must use expressions as placeholders. The {} brackets match the dBegin and dEnd fields, and

the 00000.00 matches the nAmt field in TRANX. Given this condition, the two result sets of the queries match and we combine all the individuals with and without transactions into one final result set.

One final note about this outer join methodology: SQL always matches empty records and will place those records in the result unless you guard against it by eliminating empty records with an additional WHERE clause, like so:

```
select ;
  cNaf      as DEPOSITORF, ;
  cNal      as DEPOSITORL, ;
  dBegin    as BEGIN_DATE, ;
  dEnd      as END_DATE, ;
  nAmt      as TRANS_AMT ;
from TRANX, IND ;
where TRANX.iidind = IND.iidind ;
  and not empty(dBegin) ;
union all ;
select ;
  cNaf      as DEPOSITORF, ;
  cNal      as DEPOSITORL, ;
  {}        as BEGIN_DATE, ;
  {}        as END_DATE, ;
  00000.00  as TRANS_AMT ;
from IND ;
where iidind not in ;
  (select distinct iidind from TRANX) ;
  and not empty(dBegin)
```

Now that I've discussed the old way it was done, you'll undoubtedly have a much fuller appreciation for the outer join syntax. Before getting into the nuts and bolts of all the keywords, here's what a simple outer join looks like:

```
sele ind.cnaf, ind.cnal, tranx.dBegin, dEnd, nAmt ;
  from tranx ;
    full outer join ind ;
      on tranx.iidind = ind.iidind
```

The result of the outer join is shown in **Figure 4.2**.

Note that there are six records—Al's one transaction, Bob's three transactions, and two more—one for Carla, even though she didn't have any transactions, and one for the orphan transaction. If a regular join had been performed here, the result set would have had only four records: one for Al and three for Bob. The IND record for Carla and the TRANX record for "Orphan Transaction" would have both been left out because they didn't have corresponding matches in the other table.

The figure displays two separate Query windows, each showing a grid of data. The top window has columns Cnaf, Cnai, and Cdesc. The bottom window has columns Cnaf, Cnai, Dbegin, Dend, and Namt. Both windows show rows for Al, Bob, and Carla, along with a NULL row. The data in the bottom window includes dates and monetary values.

Cnaf	Cnai	Cdesc
Al	Aggressive	Groceries
Bob	Belligerent	Widgets
Bob	Belligerent	Movie Tickets
Bob	Belligerent	New Muffler
Carla	Courteous	NULL
NULL	NULL	Orphan Transaction

Cnaf	Cnai	Dbegin	Dend	Namt
Al	Aggressive	09/18/98	09/23/98	100.00
Bob	Belligerent	06/10/98	06/20/98	200.00
Bob	Belligerent	03/02/98	03/17/98	34.00
Bob	Belligerent	11/22/97	12/12/97	250.00
Carla	Courteous	NULL	NULL	NULL
NULL	NULL	04/02/98	04/03/98	99.95

**Figure 4.2.** The resulting outer join between two tables.

While technically correct, we can set this up to look better. Here's the same outer join with friendly replacements for .NULL., using the NVL function:

```
select nvl(ind.cnaf, "No first name"), nvl(ind.cnal, "No last name") , ;
  nvl(tranx.cdesc, "No transaction") ;
  from tranx ;
    full outer join ind ;
      on tranx.iidind = ind.iidind
```

Let's examine the syntax:

```
select <field> from table1 join table2 on t1.field = t2.field
```

You'll see that there is no WHERE clause at all—it's now been relegated to simply performing filtering, as it should have been all along. (WHERE will still function as a tool to join two tables, preserving the functionality of legacy code, but such a use is now outdated.)

The new keyword is JOIN, and it has four variants:

- JOIN (inner join)
- RIGHT JOIN (right outer join)
- LEFT JOIN (left outer join)
- FULL JOIN (full outer join)

Using JOIN without a preceding qualifier simply performs a join between two tables in the same way that WHERE used to—the result is an “inner” join, one where records without matches are ignored. The qualifiers “RIGHT” (or “RIGHT OUTER”), “LEFT” (or “LEFT

OUTER”), and “FULL” (or “FULL OUTER”) now allow you to dictate how records without matches are handled.

The FULL OUTER JOIN is obvious—records without matches on both sides of the join are also included. However, the RIGHT and LEFT might not be as intuitive. There may be times when you only want matches from one side of the relationship—for example, individuals with, and without, transactions—but you don’t want orphaned transactions (those that somehow were entered or kept in the database without a corresponding individual record).

The following SELECT produces all individuals, with and without transactions:

```
select nvl(ind.cnaf, "No first name"), nvl(ind.cnal, "No last name"), ;  
      nvl(tranx.cdesc, "No transaction") ;  
  from ind ;  
    left outer join tranx ;  
  on tranx.iidind = ind.iidind
```

The following SELECT does the opposite: transactions with and without individuals:

```
select nvl(ind.cnaf, "No first name"), nvl(ind.cnal, "No last name"), ;  
      nvl(tranx.cdesc, "No transaction") ;  
  from ind ;  
    right outer join tranx ;  
  on tranx.iidind = ind.iidind
```

Let’s examine these examples in more detail.

You’ll note that, along with the OUTER JOIN keyword, an ON <expression> construct is used. This logical expression describes how the two tables are joined. The qualifier of the JOIN simply describes which side will contain records that don’t have matches on the opposing side. Thus, a SELECT command like:

```
from A right outer join B
```

will grab all records from B, whether or not they have matches in A. You could produce the same effect by using a LEFT JOIN and reversing the order of the tables, like so:

```
from B left outer join A
```

I typically keep the “parent” table on the left side of the JOIN expression, and have either children or lookup tables on the right side. There’s no technical reason for doing so—I just find it easier to visualize the result set in my mind.

Now that I’ve dissected a two-table join, it’s time to move on to joins involving three or more tables. Before doing so, I need to mention that there are two fundamental rules involved in an outer join. First, you can’t reference a table in the ON clause until it’s been opened in a JOIN. While this can’t happen in a two-table join constructed as shown in the preceding example, it’s easy to do in joins involving three or more tables, and I’ll come back to this rule. The second rule is that each JOIN/ON combination creates an intermediate result, which is then used in the rest of the JOIN.

To illustrate these rules, suppose you were joining a parent table, A, with two lookup tables, T1 and T2, both of which had foreign keys in A. You would need to define the A - T1 join, and describe the ON condition for A - T1, before getting A involved with T2, like so (I’ve

eliminated the field list and the actual field names from the ON conditions for simplicity's sake):

```
from A join T1 on A = T1 ;
join T2 on A = T2
```

Syntax like this wouldn't work:

```
from A join T1 join T2 :
on A = T1 on A = T2
```

The result of the A - T1 join couldn't be joined with T2 because the A - T1 ON condition hadn't been found by the SQL parser.

With that theory behind us, let's look at a typical use for an outer join: including the values from a lookup table in a join, whether or not the parent table always used those values.

As an example, suppose our individual table recorded information such as personality type, hair color, and type of car driven. However, instead of storing values like "BLONDE" and "CLUNKER" in the IND table, these values were stored in a lookup table, and the IND table simply held foreign keys to those lookup values. Furthermore, suppose that, instead of keeping many lookup tables lying around, each of these lookup values was stored in the same table (named ITLOOK), as shown in **Figure 4.3**.

**Ind**

lidind	Cnaf	Cnal	Ctypepers	Ctypecar	Ctypehair
1	Al	Aggressive	A	I	
2	Bob	Belligerent	B		N
3	Carla	Courteous	C	I	Y

**Itlook**

Type	Cdesc
A	Type A
B	Wimp
C	Too Mellow
1	\$1,000,000
2	\$200,000
3	\$49.99
R	Brunette
L	Black
N	None
X	Blond
Y	Blonde
O	Orange/Red

**Figure 4.3.** Sample table layouts for a lookup table.

The goal is a list of the individuals, together with their personal attributes, but a simple join won't work because some of the attributes are missing. As with many things, it's best to start slowly and build up: If you try to create a five-table join with a variety of qualifiers, you're dooming yourself before you begin.

```
select ind.cnaf, ind.cnal, itlook.cdesc ;
from ind full outer join itlook ;
on ind.ctypepers = itlook.ctype
```

This is pretty ugly—a bunch of .NULL.'s floating around. Try again:

```
select nvl(ind.cnaf, "No first name"), nvl(ind.cnal, "No last name"), ;
itlook.cdesc ;
from ind full outer join itlook ;
on ind.ctypepers = itlook.ctype
```

This still isn't what you want, because it will include records for every value in the lookup table, whether or not they are used. Clearly, a LEFT JOIN will just grab the individuals:

```
select nvl(ind.cnaf, "No first name"), nvl(ind.cnal, "No last name"), ;
itlook.cdesc ;
from ind left outer join itlook ;
on ind.ctypepers = itlook.ctype
```

However, this isn't enough—you want all of the attributes. Getting all three lookup values in one step is going to be too complex, so I'm just going to address a second attribute for now.

To do so, you have to grab personality types from the lookup table and join them with the IND table's cPersType field, and at the same time, grab car types from the same table and join them with the IND table's cCarType field. This requires a bit of sleight of hand, which involves opening the lookup table more than once, with different names each time. The p-code will look like this (again, I've simplified it by taking out the field list and field names in the ON condition):

```
from A join LOOKUP LOOK1 ;
on A = LOOK1 ;
join LOOKUP LOOK2 ;
on A = LOOK2
```

You'll see that the reference to the lookup table is followed once by the expression LOOK1, which is the alias that will be used in that join, and then by the expression LOOK2, which is the alias that will be used in the second join. Now that you see what the basic idea is, here's the real code:

```
select nvl(ind.cnaf, "No first name"), nvl(ind.cnal, "No last name"), ;
look1.cdesc as perstype, look2.cdesc as cartype ;
from ind join itlook look1 ;
on ind.ctypepers = look1.ctype ;
join itlook look2 ;
on ind.ctypecar = look2.ctype
```

The results for this and the next two queries are shown in **Figure 4.4**. This first example won't perform quite as expected; because it's a simple join, only records that exist in both the IND table and the ITLOOK table (referenced by LOOK1 and LOOK2) will end up in the result set.

```
select nvl(ind.cnaf, "No first name"), nvl(ind.cnal, "No last name") , ;
look1.cdesc as perstype, look2.cdesc as cartype ;
from ind left join itlook look1 ;
on ind.ctypepers = look1.ctype ;
join itlook look2 ;
on ind.ctypcar = look2.ctype
```

This one doesn't do much better, because it's only doing one LEFT JOIN between IND and LOOK1—the join with LOOK2 (for the car type) is still a plain old inner join. Only those IND records that have matching records in LOOK2 will be found.

```
select nvl(ind.cnaf, "No first name"), nvl(ind.cnal, "No last name") , ;
look1.cdesc as perstype, look2.cdesc as cartype ;
from ind left join itlook look1 ;
on ind.ctypepers = look1.ctype ;
left join itlook look2 ;
on ind.ctypcar = look2.ctype
```

Here's where you will strike gold—selecting every record from the IND table, whether or not there is a matching record in ITLOOK.

Query1				
	Exp_1	Exp_2	Perstype	Cartype
▶	Al	Aggressive	Type A	\$1,000,000
	Carla	Courteous	Too Mellow	\$1,000,000

Query2				
	Exp_1	Exp_2	Perstype	Cartype
▶	Al	Aggressive	Type A	\$1,000,000
	Carla	Courteous	Too Mellow	\$1,000,000

Query3				
	Exp_1	Exp_2	Perstype	Cartype
▶	Al	Aggressive	Type A	\$1,000,000
	Bob	Belligerent	Wimp	NULL
	Carla	Courteous	Too Mellow	\$1,000,000

**Figure 4.4.** Results of various lookup queries.

Note that the first two SELECTs didn't work, even though all three individuals had matches in the lookup table for personality type. Bob didn't show up because he didn't have a match in the Cartype field.

It's time to add the third lookup table. Instead of just giving you the “answer,” let's look at several possible permutations and discuss why they won't work. The following SELECT command will produce the error “Alias LOOK1 not found” because the first ON clause references both IND and LOOK1:

```
select ind.cnaf, ind.cnal, ;
ind.ctypepers, look1.cdesc as perstype, ;
```

```

ind.ctypehair, look3.cdesc as hairtype, ;
ind.ctypecar, look4.cdesc as cartype ;
from itlook look3 right outer join ;
    itlook look1 right outer join ;
        ind left outer join itlook look4;
on look1 ctype = ind.ctypepers ;
on look3 ctype = ind.ctypehair ;
on look4 ctype = ind.ctypecar

```

This next attempt produces the error “Alias LOOK3 not found” because LOOK1 and LOOK4 have been moved, but the second ON clause references LOOK3 and IND:

```

select ind.cnaf, ind.cnal, ;
    ind.ctypepers, look1.cdesc as perstype, ;
    ind.ctypehair, look3.cdesc as hairtype, ;
    ind.ctypecar, look4.cdesc as cartype ;
from itlook look3 right outer join ;
    itlook look1 right outer join ;
        ind left outer join itlook look4;
on look4 ctype = ind.ctypecar ;
on look3 ctype = ind.ctypehair ;
on look1 ctype = ind.ctypepers

```

Moving LOOK3 and LOOK1 so that the third ON clause references LOOK3, which is opened last, makes everything click:

```

select ind.cnaf, ind.cnal, ;
    ind.ctypepers, look1.cdesc as perstype, ;
    ind.ctypehair, look3.cdesc as hairtype, ;
    ind.ctypecar, look4.cdesc as cartype ;
from itlook look3 right outer join ;
    itlook look1 right outer join ;
        ind left outer join itlook look4;
on look4 ctype = ind.ctypecar ;
on look1 ctype = ind.ctypepers ;
on look3 ctype = ind.ctypehair

```

Another type of multi-table outer join involves many-to-many relationships. For example, table A can have zero or more matching records in table B, and the same goes for B—it can have zero or more matching records in table A. To accommodate this requirement, a third table, AB, is used to hold keys for both tables for every instance of a match. Getting an outer join to work in this situation is a little tricky.

Suppose you have two tables, PART and IMAGE. The PART table has records for every part stocked in the machine shop, and IMAGE has records for every blueprint in the machine shop. A single PART might not have a blueprint (because it's purchased from outside, or because the blueprint has been lost). A PART might also have a single blueprint, or it might have multiple blueprints if it's a complex part. Furthermore, the same blueprint might be used for more than one part (two parts might differ only in material, color, or dimension, thus allowing the same blueprint to be used for each). Thus, the PARTIMAGE table contains keys from both the PART and IMAGE tables.

In this example, all records in PART will show up in the result set. Remember that the join table PARTIMAGE might have zero, one, or more IMAGE records for any specific record in

PART. Thus, for a given record in PART, the result set might have a single record populated with .NULL. (indicating that there is no match in IMAGE—no blueprint exists), or one or more records from PART, indicating one or more matches (blueprints) for that part.

```
select ;
  nvl(cNoPart, "NO PART") as PartNumber, PART.cdepm as PartDesc, ;
  nvl(IMAGE.cnaimage, "NO IMAGE JACK") as ImageName ;
  from IMAGE full outer join PARTIMAGE ;
    on PARTIMAGE.iidimage = IMAGE.iidimage ;
    full outer join PART ;
      on PART.iidpart = PARTIMAGE.iidpart
```

## Controlling the result set: destination

So far we've just worked with producing a result set, and "let the records fall where they may." However, we often want to do something special with the results of our query. For instance, we might want to order the result set in a particular way or use a second SELECT command on the contents of the result set.

You can send the results of a SELECT command to several different destinations. As you've seen, by default, Visual FoxPro uses a browse window named Query as the destination. You can specify one of three destinations for the output of a query: an actual table, a temporary table (located in memory) called a cursor, or an array.

The first destination is an array. This is useful for small result sets, such as a lookup table that will populate a combo box or list box on a form. It is also useful when you are using a SQL command to perform an aggregate function such as COUNT or SUM. More overhead and maintenance is necessary to place this information into a table or a cursor, and its size doesn't impact memory to any practical significance.

You need to remember that if the result set is empty, the array won't be created at all. You can use the \_tally system variable to determine if the result set was empty. For instance, the following program code will place a value into the memory variable nTranSum, regardless of whether there were any qualifying records:

```
select ;
  sum(nAmt) ;
  from TRANX ;
  where between(dBegin, {1/1/94}, {12/31/94}) ;
  into array aTranSum
if _tally = 0
  m.nTranSum = 0
else
  m.nTranSum = aTranSum[1]
endif
```

It looks like a bit more work, but using this construct as a standard practice will save you a lot of headaches and debugging in your programming.

The second destination is a table. If the result set of the query is going to be large (to preclude the use of an array) or if you need the result set to "hang around awhile," you'll want to send the result set to a table. This option actually creates a .DBF file (a free table) on disk that can be edited, modified, and otherwise treated like any other table. Note that this table will

be created without warning if the setting of SET SAFETY is OFF, regardless of whether or not the table already exists, or whether it's even open in another work area.

The third destination is a cursor. We've been using the term "result set" to refer to the collection of records created by the execution of a SELECT command. While Visual FoxPro is "record oriented" in that we can move from record to record as we desire, other languages that use SQL do not have this capability and can only address a block of records as a single entity. As a result, all SQL operations are oriented toward manipulating this "set of records." They refer to this block of records as a "cursor," which stands for CURrent Set Of Records. A cursor is a table-like structure held in memory; it overflows to disk as memory is used up. In some cases, such as queries that essentially just filter an existing table (in other words, no calculations or joins), Visual FoxPro opens a second copy of the table in another work area and then applies the filtering condition to present a different view of the table as the query requests it.

Cursors have two advantages. First, they are fast to set up, partly because of the possible filtering of an existing table, and also because of their initial creation in RAM. Second, maintenance is greatly reduced, because when a cursor is closed, any files on disk created due to RAM overflow are automatically deleted.

## Controlling the result set: ORDER BY

Records in a result set are presented in random order unless you use the ORDER BY keyword to control the order. Unlike indexes, which present a different view of the records but don't change the physical position, ORDER BY actually positions the records in the order specified by the ORDER BY field list.

You can use field names or the number of the position of the column in the field list. The latter option is useful when using functions that create columns with an unknown name. The default order is ascending, but you can override it by using the DESCENDING keyword.

Here's how to sort a result set by "transaction-begin date" and then by transaction amount within a single transaction-begin date:

```
select ;
  cDesc      as DESCRIPTION, ;
  dBegin    as BEGIN_DATE, ;
  dEnd      as END_DATE, ;
  nAmt      as TRANS_AMT ;
from TRANX ;
order by dBegin, nAmt
```

To sort the result set of a GROUP BY SELECT command by the summed amount, use the column number of the aggregate function:

```
select ;
  iidind, ;
  count(nAmt), ;
  sum(nAmt) ;
from TRANX ;
where between(dBegin, {1/1/94}, {12/31/94}) ;
group by iidind ;
order by 3
```

## Controlling the result set: DISTINCT

In cases where we are selecting a subset of the available fields in the source (either table or tables), it is possible to end up with a number of identical records. For instance, the Company table might contain multiple branches of the same company with different addresses but the same company name. If we want a list of all the companies in the table, we wouldn't necessarily want multiple copies of the same company name. The DISTINCT keyword eliminates all but one instance of duplicate records in the result set.

This is an important point: The DISTINCT keyword operates on the result set, which means just those fields in the result set, regardless of whether or not other fields in the original record would make two records in the result unique. This can be handy for determining values to use to populate a lookup table, for instance. Another reason to use the DISTINCT keyword is to find instances of typos in free-form text fields. For example, if the user has typed “Visual FoxPro”, “Visaul Foxpro”, and “Visual PoxFro” in a field, a SELECT command of this sort would identify those errors. You would need to use a second SELECT command to find those records containing “Visaul FoxPro”, of course. The syntax of the command to create a list of all types of transactions from the TRANX table would look like this:

```
select distinct ;
  cDesc as DESCRIPTION ;
  from TRANX
```

In many cases, you will find you can use a GROUP BY SELECT command in place of the DISTINCT keyword—it's considerably faster. The same query with a GROUP BY clause would look like this:

```
select cDesc as DESCRIPTION ;
  group by cDesc;
  from TRANX
```

## Controlling the result set: HAVING

The SQL SELECT command has another keyword that many programmers confuse with the WHERE keyword: HAVING. While it sounds like it would perform the same function, and, in fact, will do so, there is an important difference between WHERE and HAVING. While WHERE operates as a filter on the table to produce a filtered result set, HAVING operates on the result set to filter out unwanted results. For example, suppose the result set consisted of subtotals of transactions for multiple individuals, but we only wanted to see the individuals with high subtotal numbers. HAVING would enable us to weed out the “low subtotal” folk, like so:

```
select ;
  count(nAmt) ,
  sum(nAmt) ;
from TRANX ;
where between(dBegin, {1/1/94}, {12/31/94}) ;
group by cIDIN
having sum(nAmt) > 1000
```

Note that the function or expression in the HAVING clause does not have to be the same as in the fields list. For instance:

```
select ;
  count(nAmt) ,
  sum(nAmt) ;
from TRANX ;
where between(dBegin, {1/1/94}, {12/31/94}) ;
group by iidind
having avg(nAmt) > 640
```

Again, HAVING will produce the same results as WHERE, but it is not Rushmore optimizable and, as a result, will take considerably longer than WHERE in any but the most trivial of SELECTS. As a result, you should limit your use of HAVING with GROUP BY selects, because HAVING will work on the result set, but WHERE won't. HAVING, as opposed to WHERE, gives you the ability to winnow the results of a GROUP BY SELECT command.

## INSERT

Many old-timers are used to using the APPEND BLANK command to add a record to a table. The same operation is considerably easier and more efficient with the SQL INSERT command. As opposed to the APPEND BLANK/REPLACE or APPEND BLANK/GATHER command combinations that make two passes at the table—one to add the record, and a second to update the data in the blank record—INSERT does both in one pass. For example:

```
* create variables that contain data that will be
* placed into the new record
*
* this example has a new field for middle initial
*
store "David" to cNameFirst
store "L." to cNameMiddle
store "Bowie" to cNameLast
```

```
store {4/16/47} to dBirth
*
* use SQL INSERT to add and update the record at the same time
*
insert into IND from memvar
```

Notice that, as long as IND has fields that match up with memory variables, the INSERT command does not have to be changed when the table structure changes. The INSERT command also has variations that allow you to take into account inserting specific values for specific fields, and inserting values from an existing array instead of from memory variables. You might use the first instance when you want to add a record with data for only certain fields:

```
insert into IND (iidind, dLastMod) values (1, date())
```

I always format an INSERT INTO command like so:

```
insert into IND ;
(iidind, dLastMod) ;
values ;
(1, date())
```

Unless you have a very small number of fields, you'll have to break the command somewhere, and I have found keeping the field list and values list on separate lines makes it easy to line up each matching field and its corresponding value.

Two common mistakes that send many programmers back to the Language Reference or online help are including the FIELDS keyword in the INSERT INTO command when it's not necessary, like so:

```
insert into IND ;
fields ;
(iidin, dlastmod) ;
values ;
(1, date())
```

or forgetting the parentheses around the field list or values list. If you do either of these things, VFP will respond with a Syntax Error message.

Don't feel too bad if you do it once or twice—I still do it after years of writing INSERT INTO commands. When inserting values from an array, it's important to note that the array must be structured in the same order as the table. If you don't have data for certain fields, you must still provide empty columns in the array as placeholders for those fields:

```
decl aNames[1,4]
aNames[1,1] = "Herman"
aNames[1,2] = ""
aNames[1,3] = "Munster"
aNames[1,4] = {7/4/44}
insert into IND from array aNames
```

INSERT INTO will add one record for each row in a multi-row array:

```
decl aNames[2,4]
```

```
aNames[1,1] = "Herman"  
aNames[1,2] = ""  
aNames[1,3] = "Munster"  
aNames[1,4] = {7/4/44}  
aNames[2,1] = "Lilly"  
aNames[2,2] = "O."  
aNames[2,3] = "Munster"  
aNames[2,4] = {7/4/48}  
insert into IND from array aNames
```

## DELETE

Just as you can add records to a table using a SQL command, you can delete them by using the DELETE command. A WHERE clause is used to define which record or records are to be deleted, like so:

```
delete from IND ;  
where cNaL = "Munster"
```

As with the Xbase command, DELETE, the records are just flagged for deletion—they’re not actually removed until you use the PACK command. You can also remove the delete flag from a record by using the RECALL command. Remember that RECALL is not a SQL command—you’ll use a FOR clause, not a WHERE clause with it!

## UPDATE

The UPDATE command allows you to change values in a single table, using both a WHERE clause and a SET clause that identify the field to be changed and the value to be used. For example, if you wanted to convert all of the first and last names in the IND table to uppercase, you could use a single UPDATE command like so:

```
update IND ;  
set cnaf = upper(cnaf) , ;  
cnal = upper(cnal) ;  
where cNaL <> "Mac Donalds"
```

There are a couple of limitations. For example, you can update data in only one table at a time, although that table doesn’t have to be in the currently selected work area. Unlike REPLACE, which locks the whole table, UPDATE uses record locking to make modifications. This means you can use this command in multi-user situations more easily, but because each record has to be individually locked, performance isn’t as good as using REPLACE to update a large number of records. Finally, don’t confuse this command with the Xbase command UPDATE, which is old and should never be used anymore.



## **Section II**

# **Programming Tools**

Now that you're done with Section I, you've got all the tools you need to use Visual FoxPro interactively. It's time to learn about the tools you'll use to build Visual FoxPro applications, including the following: the Editor, which allows you to create programs (standard text files that are compiled and executed in an interpretive fashion); the Menu Builder, which allows you to create Windows-standard CUA menus; the Form Designer, which you will use to create the user interface to your applications; the Class Designer, the mechanism for creating and managing your object-oriented components; and the Report Writer, the tool you'll use to produce output for your applications.



# Chapter 5

## Creating Programs

**Because an application, in its barest form, is just a program, it's appropriate to begin by creating programs. In this chapter, I'll discuss the Visual FoxPro editor, how to create and run a program, and how to customize the editing environment to suit your particular preferences.**

A Visual FoxPro program is a text file that consists of one or more VFP commands or functions. You can create this text file in any word processor or text editor that can produce an ASCII text file. In previous versions of FoxPro, many developers did this in order to take advantage of features that weren't then available in the Fox editor. Starting with Visual FoxPro 5.0 and continuing today, the VFP editor has been enhanced to the extent that there's no reason not to use it, and a number of good reasons to do so.

First, you need to be in the VFP IDE anyway to run and compile your program. Second, the editor is integrated into the environment; even with Windows, it's awkward to switch between an external editor during editing and Visual FoxPro to run, test, debug, and compile the file. Third, a proper VFP application typically has very little program code—most of the application will be found in forms and classes, and the code in these components must be edited in the VFP environment.

I'd like to emphasize this last point. Although this chapter will concentrate on programs, this is merely to make you comfortable with the editor and the process of writing blocks of code. While applications in the pre-GUI environment relied largely on program files, you'll primarily be using the visual tools, with blocks of code contained in components built with those tools, to build your applications.

### Creating a program with the editor

The Visual FoxPro editor produces a text file that automatically has an extension of .PRG. This file is compiled into pseudocode, and the resulting file, having the same name as the .PRG file but an .FXP extension, is interpreted by the Visual FoxPro engine. If you run this program from within VFP, the development environment has the interpreter automatically available. If you create a stand-alone executable, you'll need to have the VFP runtime files available to the executable so that the engine can interpret the executable.

To create a program file, issue this command in the Command window:

```
modify command
```

Or open the File menu and select New, Program, New File. An empty text-editing window will appear with the title PROGRAM1. If you like, you can specify the name of the file by entering this command in the Command window:

```
modify command xxx
```

An empty text window with the title XXX.PRG will display, where XXX.PRG is the name of the program.

After typing something in the window, you can save the program by opening the File menu and clicking Save, or by pressing Ctrl+S. If you've not already named the file, the Save As dialog will appear, prompting you for a file name. Note that the file will be given a .PRG extension automatically. You can override this extension, but Visual FoxPro will not recognize the file as a program and you will have to include the extension as part of the file name each time you want to edit or run it, so it's a good idea to use Visual FoxPro's .PRG default.



*The file will be saved in the default directory. You can determine the default drive and directory with the SYS(5) and SYS(2003) functions, or determine just the default directory with the CURDIR() function or CD() command. You can change the default directory for the current FoxPro session with the SET DEFAULT TO <dir name> or CD <dir name> commands. Note that the File/Open dialog box displays the last directory that the file open dialog pointed to—not necessarily the default directory.*

```
? "The default drive and directory are: " + sys(5) + sys(2003)
? "The default directory is..."
cd
```

To create a Visual FoxPro program, type the Visual FoxPro commands in the text file, each on a separate line. You can separate commands with blank lines and indent commands using either spaces or tabs for better readability. Lengthy commands can be continued on succeeding lines by ending the line to be continued with a semicolon. Comment lines are created by placing an asterisk as the first character of the line. You can include comments on the same line as non-continued commands by preceding the comment text with double ampersands. And here's a point that catches most everyone sooner or later: The Visual FoxPro continuation character—the semicolon—also works on comment lines and lines that end in an && style comment. Here's an example:

```
*      Program : XXX.PRG
*      Author : Whil Hentzen
*  Description: Sample program to show types of comments
*                  : This program increments a number in one of two ways
*                  : according to whether the number is even or odd

x = 1                      && initialize counter
for j = 1 to 10
*
* increment x differently according to
* whether it is odd or even
*
x = x + ;
    iif( x/2 - int(x/2) = 0, ,
        x + j, ;
        x + 2 * j )
*
* display the results of x to the screen
*
```

```
? x
endfor

* warning: the next line is regarded as part of this comment ;
return

* many older programs use the next line to visually indicate
* the end of the program file
* <EOF>
```

The last line—the comment that marks the end of the file—is strictly an aid to the programmer. It is not needed by Visual FoxPro, and indeed is regarded by many as an anachronism. In the old days, some Xbase variants couldn’t handle files past a certain length, but instead of prohibiting you from creating a file that long, the file would simply be truncated when saved. As a result, developers used defense mechanisms like a visual End of File marker. Some still use it today for ease of determining the end of a program in printed output, but the choice is up to you.

## Running a program

All C programming books start out with the same program that displays the phrase “Hello World” on the screen, and we can do something similar for our first program here. First, open a new program file named HELLO.PRG:

```
modify command HELLO
```

If you want to use a file name that contains spaces, you need to enclose the file name in quotes, like you would elsewhere in Windows:

```
modify command "HELLO WORLD"
```

Next, type the following code into the window:

```
*      Program : HELLO.PRG
*      Author : Whil Hentzen
*      Description: Hello World sample program

messagebox("Hello World. Press Any Key to Continue...", 0 , "My First VFP
Program")

return

* <EOF>
```

Once you have typed in your program and saved it, you can compile and run it in one step. You can right-click in the text-editing window and select the DO HELLO.PRG menu command. If the text-editing window that you typed your program in is the active window, the last menu option in the Program menu will contain the command DO, together with your program name. You could also just press Ctrl+E. After you perform one of these actions, Visual FoxPro will compile the text file into p-code, give the resulting file the name of the .PRG file but with an .FXP extension, and then run the resulting file. A third method is to save the program and then enter the DO XXX command in the Command window, where XXX is the

file name. Note that you do not have to include the extension if you saved the file with a .PRG extension. If you haven't saved your program yet, you will be prompted to do so before the program is compiled and run. Go ahead and run your program now if you haven't already.

You'll see a standard Windows message box containing the message "Hello World. Press Any Key to Continue ...". You'll also notice that the Command window will have disappeared. The message box command is being executed and has halted program execution until you respond to the message box. To continue, close the message box by clicking OK or clicking the close box in the title bar. The message box will go away and you'll be returned to interactive Visual FoxPro.

## Compiling a program

When you execute a program from within the VFP IDE, Fox will automatically compile the .PRG to an .EXP. You can perform this operation manually by opening the Program menu and selecting Compile to create the .EXP file, and then selecting the Do menu option from the same menu.

As long as you are using the Visual FoxPro editor, each time you make changes to a program, Visual FoxPro will automatically compare the date and time of the .EXP file of the same name and, if the .PRG file is newer, recompile the program before running.

You can also create three other types of "compiled" files: an .APP file, an .EXE file, and a COM .DLL file. An .EXE is a file that can include one or more .PRGs (as well as other components, such as forms and reports) that can be run both within and without the VFP interactive environment. For example, you can attach a Windows desktop shortcut to a VFP-created .EXE, and run the application just like any other program. You can also use the Start/Run command, or call it from another program as you would any other .EXE. The only constraint is that the VFP runtime files (plus any other files required by the application) must be in the path searched by the .EXE file.

An .APP file is halfway between an .EXP and an .EXE. Like an .EXE it can include one or more components, but it can't run independently of the IDE. However, you can create an .EXE that then calls an .APP file; this technique is often used to distribute modules of an application without redistributing the entire system.

A .DLL file is a COM component, or, as it used to be known, an automation server. VFP 5.0 introduced the ability to create stand-alone COM components that could be called by other applications, similar to (but not exactly like) COM components built in Visual Basic or C++.

To produce any of these, you need to use the Visual FoxPro Project Manager, which I'll discuss in Chapter 6.

## Up close with the VFP editor

Before Visual FoxPro 5.0, about all the flexibility you had with the editor was the ability to change the size of the editing window. Not terribly robust, but things have changed significantly since then. Because some of you are coming from FoxPro 2.x or VFP 3.0 environments, it's time to discuss the new features. If you've already used the editor in 5.0 or later, you can skim this section quickly.

The Visual FoxPro editor behaves just like all other Windows editing programs. It offers complete cut, copy, and paste functionality between multiple Visual FoxPro editing windows as well as other Windows programs.

As you begin typing commands into a program file, you'll see that the various pieces are color coded, just as they are in the Command window. You can control which colors are used for which components by opening the Tools, Options dialog and selecting them from the Syntax Coloring tab. See **Figure 5.1**.

You can control coloring for the following types of components:

Comments	Any line that begins with an asterisk or the NOTE keyword, or the rest of the line, starting with the “ <b>&amp;&amp;</b> ” comment designator.
Keywords	All Visual FoxPro keywords. I've found this useful to make sure I'm not using little-used keywords as names for files, classes, field names, or indexes.
Literals	A value like a number, such as 3.14159.
Normal	All text entered into a file that is not a program, form, or class—for example, if you just created a text file or were viewing a directory listing that had been output to a file.
Operators	Characters such as addition and subtraction signs, equal signs, and periods when used to delineate memory variables or objects' properties and methods.
Strings	Text strings enclosed in single or double quotes, or square brackets.
Variables	Standard memory variables as well as method and property names.

If you click the Set as Default button, these settings are saved in the Windows registry.

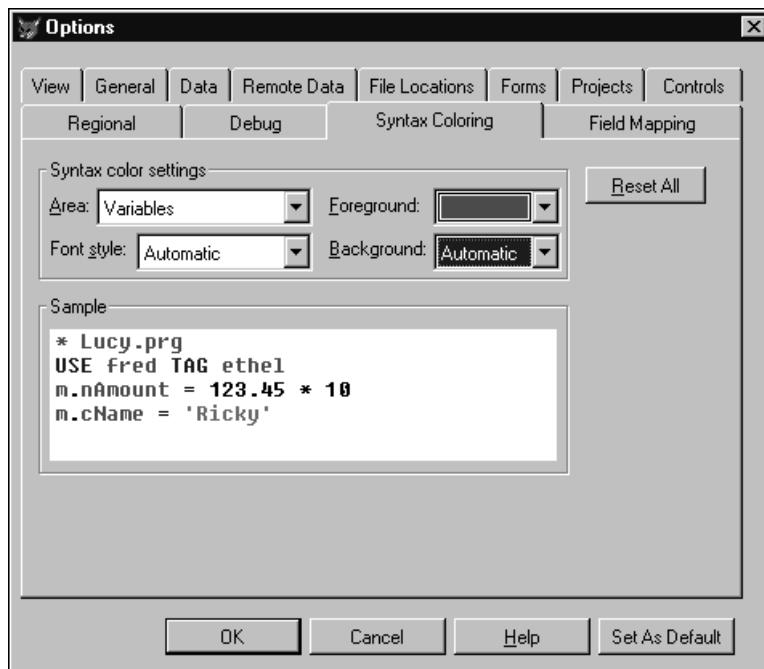
If you right-click in an open text-editing window, you'll see a variety of available options. The first three—Cut, Copy and Paste—are also available through other means. The Build Expression option brings forward the Visual FoxPro Expression Builder or the expression builder you've specified in the File Locations tab in the Tools, Options dialog.

You can highlight one or more lines of code in a program file, and then select the Execute Selection menu option and VFP will execute each line of code sequentially. This is particularly handy to do from within the Command window, since it saves time having to copy several lines of code to a temp file and then run that temp file. However, it also saves time copying a block of code out of a program file to that annoying little temp file that will end up cluttering your hard disk.

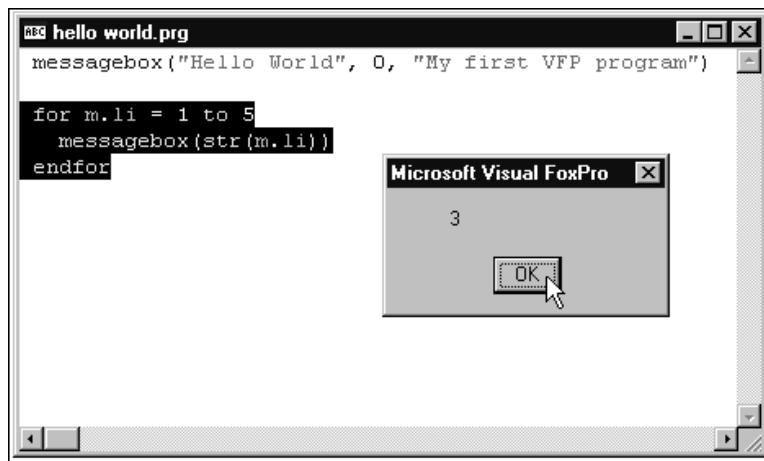
A little-known trick is that you can execute code in a loop by using the Execute Selection command. For example, enter the following code in a program file:

```
for m.li = 1 to 5
  messagebox(str(m.li))
endfor
```

Then highlight the entire block and select Execute Selection from the context menu. You'll be greeted with a series of five message boxes, as shown in **Figure 5.2**. This is definitely a lot easier than having to run the code in a separate program.



**Figure 5.1.** You can set which colors are used for various components in the Visual FoxPro editor by using the Tools, Options dialog.

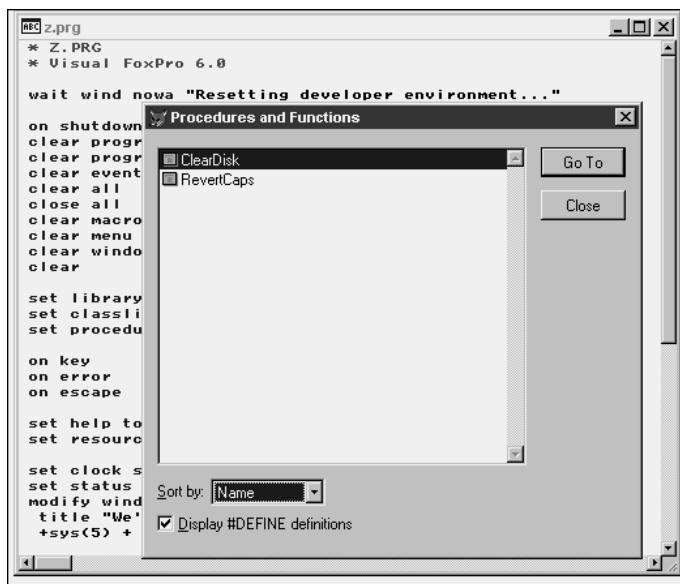


**Figure 5.2.** To run a loop of code, use the Execute Selection menu command from a program file's context menu.

You can also run the program by opening the Program menu and selecting DO Hello World.PRG.

The next menu option in the context menu is Procedure/Function list as shown in **Figure 5.3**. This function can be used to navigate through a program file and locate a specific procedure or function. This was useful in pre-OOP versions of FoxPro where developers would create a program file that contained a large number of functions, and you found yourself paging through thousands of lines of code to find the one procedure you wanted. With Visual FoxPro, that function library often takes the form of a class library, and each procedure or function is contained in its own class.

Set Breakpoint is used in debugging, and I'll cover that in Chapter 19, "Using the Debugger."



**Figure 5.3.** The Procedures and Functions dialog allows you to move to a specific procedure or function.

The Font menu option opens the Font dialog so you can change the font of the file. Note that this doesn't work like a word processor where you can highlight part of a file and change the font attributes of just that section. It's an all-or-nothing choice for a program file. This ability is provided so that you can change the characteristics in order to see the file better. I often change the font of a program file or class method to FoxFont, 7 point, so that I can see 100 lines or more of program code in a single window. Just temporarily, of course—doing so permanently would probably cause blindness.

The Find and Go To menu commands allow you to navigate through a program file in two more ways. The first command opens a standard Windows Find dialog. The Go To command allows you to enter a line number to navigate to. This is particularly handy if you've run into a program error "on line 295"—instead of hitting "Page Down" multiple times, you can go

straight to the line in question. Given its usefulness, why a keyboard shortcut wasn't provided for this operation is beyond me.

The Beautify menu option allows you to specify how the contents of an editing window will be capitalized and indented. You can control how keywords and symbols, independently, are capitalized. Keywords can be converted to uppercase, lowercase, mixed case, or left alone. Symbols can be converted to uppercase, lowercase, left alone, or converted to the same use as the first occurrence (in the current editing window). You can also choose to indent using tabs or spaces, or to leave the indenting alone. You can choose how many spaces, if you want to use spaces, and you can specify what type of text to be indented: comments and/or continuation lines. You can also choose to indent lines following the beginning of procedures and within DO structures.

Many developers tend not to use Beautify on their own code because they have trained themselves to use a certain style. However, this tool is quite useful if you've inherited someone else's code, or if you are trying to bring the work of several developers into sync.

The Indent and Unindent commands can be used on a block of code. The Indent command indents the highlighted block of code using a tab equivalent to four spaces; the Unindent command will remove four characters (or a tab character) at the beginning of each highlighted line. Note that you don't have to have used the Indent command (or even used the Tab key) on a line in order to take advantage of Unindent. This can also be done by pressing the Tab key or Shift+Tab.

The Comment command will insert a character string of “\*!\*” and then a tab before each highlighted line as shown in **Figure 5.4**. It's hard to tell that the separator is a tab because VFP's default tab character takes up four spaces, but you can verify this by deleting part of the comment string—the commented text will stay indented at the fifth position.



```
REC Z.PRG
* Z.PRG
* Visual FoxPro 6.0

wait wind nowa "Resetting developer environment..."

on shutdown
clear program()
clear program
clear events
clear all
close all
clear macros
clear menu
clear window
clear

*!*      set library to
*!*      set classlib to
*!*      set procedure to

on key
on error
on escape

set help to
```

**Figure 5.4.** You can comment a block of text in a text window with the Comment menu option in the editor's context menu.

You can change the character string that VFP uses as the default comment string by spelunking in the Windows registry, but I've found it useful to use my own set of comment characters for various purposes, and then use the VFP Comment command to identify a block of code that I want to comment out temporarily. Seeing “\*!\*” in my code is a visual reminder that I performed the programming equivalent of leaving a sponge inside the patient. You can use the Uncomment command to remove the “\*!\*” string from the beginning of a line of code, and, like the Unindent command, you don't have to have used the Comment command first. In fact, if you've just typed “\*!\*”, VFP will remove those characters with Uncomment. If you've typed “\*!\*”, VFP will leave the space, just removing the first three characters. I'm sure this is more than you ever wanted to know about VFP's commenting and uncommenting features.

The last option on the context menu is Properties, and it opens a dialog that allows you to determine a variety of properties for the current text-editing window as well as for all files. See **Figure 5.5**.



**Figure 5.5.** The Properties dialog allows you to specify a variety of attributes for a text-editing window.

If you're used to other Windows word processing or editing programs, you're probably familiar with drag-and-drop text editing. This feature is known as “drag and drop” because you “drag” the selected block to a new location, and then, by releasing the mouse button, you “drop” the selected block in its new location. Note that this operation “moves” text; it does not copy it. If you're uncomfortable with this capability, you can turn it off by removing the check from the Drag-and-Drop Text Editing check box.

For non-.PRG files, you can choose to wrap text when your typing extends past the edge of the editing window (for .PRG files, you probably don't want your program code to wrap).

Each time you press the Enter key in the editor, Visual FoxPro will insert a carriage return and line feed key combination, and it will start the next line at the same column as the previous line started. If the previous line was indented several characters, the new line will also be automatically indented the same number of characters. This feature is handy when you are typing a block of code inside a logical structure and want the entire block to be indented the same amount. If you want to turn this behavior off, remove the check from the Auto Indent check box.

These three options are all specific to the current window.

You can set a variety of options for saving a file, including whether or not to make a backup copy of it (a file with the same name but a .BAK extension), and so on. For .PRG and .TXT files, I just keep the default settings. Automatically compiling your programs before saving is a good idea because it doesn't take any noticeable time, and you'll be alerted to any errors the compiler catches before attempting to run the program. You can also specify whether or not line feeds and an end-of-file CTRL+Z character are saved with the text. Visual FoxPro's defaults can be changed if you need a specific behavior when manipulating a specific file.

The Appearance options control how the file will display in the editing window, including font attributes, whether or not to show line/column positions in the status bar, and whether to use syntax coloring. When you press the Tab key in the editor, the cursor is advanced the number of spaces specified by the number in the Tab Size spinner. The default tab value is 4. If you want to have a different indent for the Tab, change it here.



*My personal preference is to use the spacebar instead of the Tab key for indenting. Different editors and printers react differently to the Tab character, and nicely formatted text with specific tab spacing will look garbled if different tab spacing is accidentally selected. Other developers have just the opposite preference. As with many things, your mileage may vary. Do what is most comfortable for you, but be consistent about it!*

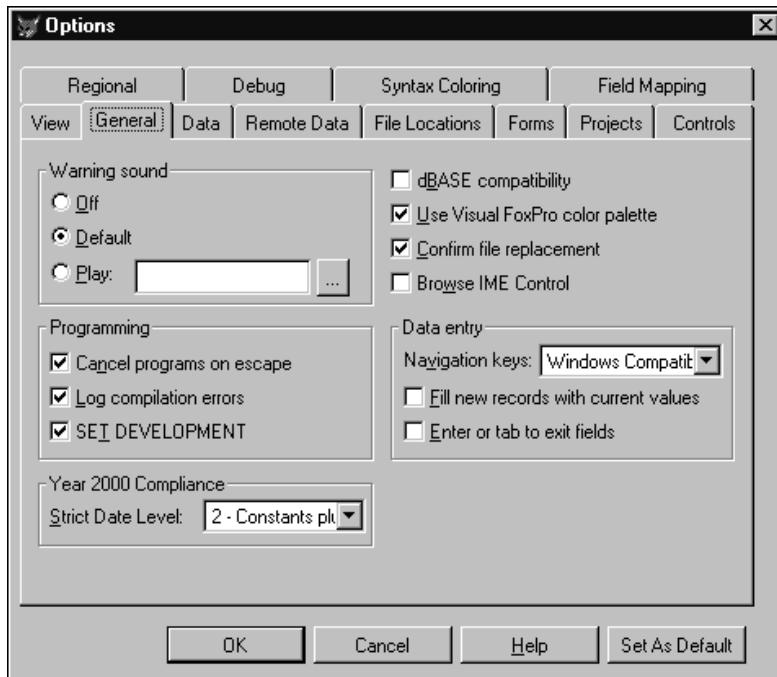
You can specify whether or not Visual FoxPro will display the current line/column position of the cursor in the Status Bar during editing by using the Show Line/Column Position check box. I always keep it on because it's handy to know what line I'm working on. Whether the line number appears is a function of whether Word Wrap is on, and that's yet another reason for keeping Word Wrap off for programs.

You might choose to turn off syntax coloring if you're editing a file that has a .PRG extension but that doesn't contain Visual FoxPro program code. For example, a file that contains #DEFINE constants could be difficult to read if syntax coloring was turned on.

You can choose to save these preferences, and to apply these settings to all files of the type that you are currently editing. These settings are saved to the Visual FoxPro resource file, not the Windows registry!

## Configuring Visual FoxPro program execution preferences

You can control the way Visual FoxPro works during program execution through options specified in the General tab of the Tools, Options dialog box. See **Figure 5.6**.



**Figure 5.6.** The General tab of the Tools, Options dialog allows you to control what happens during program execution.

Normally, you can interrupt a Visual FoxPro program by pressing the Escape key. You can prevent this from happening by removing the check from the Cancel Programs on Escape check box. This is equivalent to using the SET ESCAPE OFF command.

If the Visual FoxPro compiler catches errors when it is compiling a program file, and if the Log Compilation Errors box is checked, a file with the same name as the program but with an .ERR extension is created. This text file will contain a list of all the errors the compiler caught, together with the offending lines of code and their line numbers. If you don't correct the program, the offending line of code will be highlighted and an error message will appear.

You can also turn the SET DEVELOPMENT setting (used to control recompilation when using third-party editors) on or off as desired. This is equivalent to using the SET DEVELOPMENT ON|OFF command.



# Chapter 6

## The VFP Project Manager

**The Project Manager is Visual FoxPro's mechanism for organizing all the files that make up an application. Because a complex application can be comprised of thousands of files—tables, classes, programs, forms, reports, menus, libraries, and so on—of a dozen or more types, and even a simple application can consist of several dozen files, it's a good idea to use the Project Manager for all of your applications. It's also the only way to create executables, so even if you avoid the Project Manager during your day-to-day work, you'll need to use it to produce your finished result.**

The Project Manager enables you to view, in a variety of hierarchical structures, some or all of the files that make up an application. It allows you to automatically call up the associated editing tool for any of those files by clicking on the file name, and to create compiled .APP, .EXE, and .DLL files for easier and more secure distribution.

An application ordinarily consists of a main program file that in one way or another calls multiple programs, forms, reports, menus, and other files. Oftentimes, the main program will call a menu, and the rest of the application's functionality is accessed through menu options that call other programs, forms, and reports. Additionally, any of the files may reference other files such as graphics or icons, text files, help files, databases and tables, and so on.

You don't have to construct the entire application before using the Project Manager. In fact, one of its purposes is to help organize files as you add them to your application.

### Using the VFP Project Manager to build an .APP or .EXE

The following steps show you how to use the Project Manager to create an executable program:

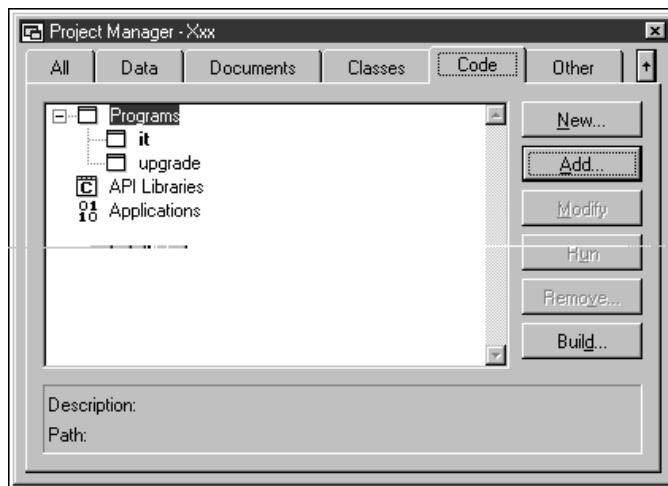
1. Open the File menu and select New, Project, New File, or type:

```
create project xxx
```

where XXX is the name of the project to create.

2. The Project Manager appears. The name of the project displays in the half-height title bar. This screen is used to perform most of the project management functions. When the Project Manager is the active window, the Project menu pad appears on the main menu.
3. Select the Code tab and the Program outline level. See **Figure 6.1**.
4. Click the New or Add button to create or add the main program file. If it's the first file added to the project, the file name will automatically be displayed in boldface. This indicates that this file is the main (or calling) file in the application. If you add the main program file later (for instance, you had a project with a main file, but then later

add another file that becomes the main file) and it doesn't automatically get marked as the main file, open the Project menu and click Set Main to do so.



**Figure 6.1.** The program file “it” is the top-level program for the Xxx project.

5. Once you have a main program file, click the Build button. The Build Options dialog will appear. Select the Build Project option button. All files referenced by the main program, either directly or indirectly, will be brought into the project under the appropriate tab and outline level.

## Up close with the Project Manager

### The Project Manager dialog

The Project Manager interface is powerful and flexible. While the functions of specific interface elements such as command buttons and the tabbed dialog work as they do in other Windows applications, some specific features of the Project Manager require explanation.

You can view all of the outline levels in one window by selecting the All tab. See **Figure 6.2**. If the Project Manager window isn't big enough, you can resize it vertically and horizontally as needed. You can also scroll through the list of outline levels and files by using the list box control.

Files that can't be included as part of an .APP or .EXE file are marked with a circle with a slash through it. A typical example is an API library. Such a file can be brought into the Project Manager and marked as “excluded” so you can see that it's part of the application but it still needs to be distributed as a separate file. All other files are defined as “included” by default. The main reason, then, for adding a file to a project when you're going to exclude it is to allow

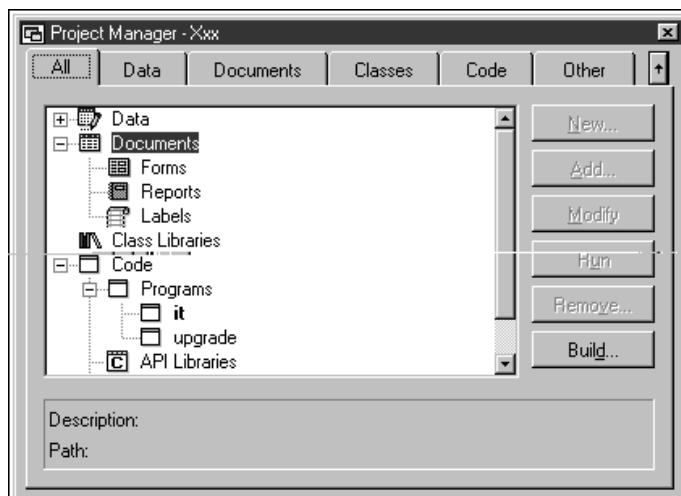
easier access to the file for viewing or editing. While you wouldn't edit an API library, you might want to edit an INCLUDE (.H) file or the system specification in a Word document.



*If you include tables or indices in a project, those files will be marked as read-only—generally not a useful capability for a data entry and reporting system but useful for static tables.*

Clicking the plus-sign icon next to an outline level (or the picture icon next to the plus sign) will expand that outline level and turn the icon into a minus sign. Clicking the minus icon (or the picture icon next to it) will contract the outline level and change the minus icon back into a plus icon. (A common mistake is to try to double-click the icon; it just requires a single click.)

This ability to expand an outline level allows you to “drill down” from level to level. This is particularly powerful in the Data level, where you can go from an entire database down to an individual field in a table. Because this drill-down list can get long, breaking out the various components of the project into their separate tabs comes in handy.



**Figure 6.2.** The plus-sign and minus-sign icons allow you to expand levels within the Project Manager.

Once you've highlighted a file, the description and fully qualified path of that file will be displayed at the bottom of the Project Manager window. You can change the Description by right-clicking the file and selecting Edit Description.

Double-clicking the name of a file will open that file in the appropriate editing tool. For instance, double-clicking a menu file will open the Menu Designer with that menu loaded. Double-clicking an embedded .APP (which is marked with the excluded symbol) will open the project that built it if the project can be found in the path. Double-clicking a file that has another application associated with it through Windows, such as a .BMP (Paintbrush) or an .XLS (Excel), will load the file in that application.

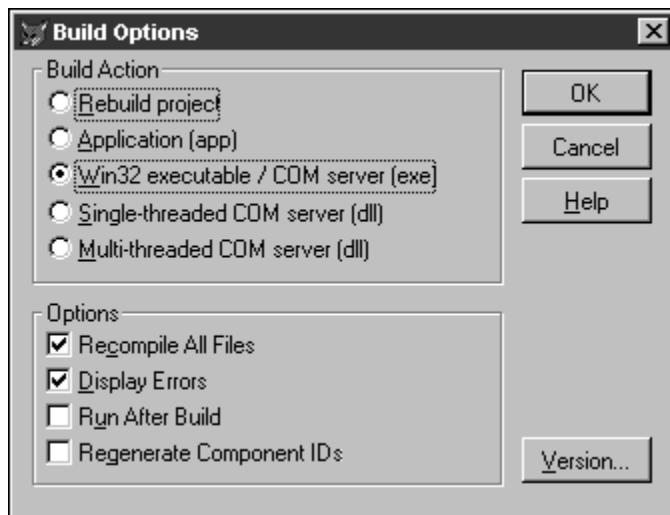
The command buttons in the Project Manager allow you to:

- Create a new file (the type depends on which outline level is highlighted in the list box)
- Manually add or remove a file
- Modify or view a highlighted file
- Execute a program
- Build (or rebuild) the project

Some of the command buttons change their label and/or purpose depending on the type of file highlighted. For example, the Run button changes to Open when the highlighted file is a database, to Browse when the highlighted file is a table, and to Preview when the highlighted file is a Report or Label. Be careful when using the Run button. If you are running a program on a lower level, menu, or form, it might not operate (or might operate incorrectly) if it relies on variables or data that are set up in a higher-level program.

The Remove button allows you to simply remove the file from the project. You can choose to just remove the file from the project or to also delete it from the disk.

The Build button allows you to build the project, or create an .APP file, an .EXE file, or a single-threaded or multi-threaded .DLL. Clicking Build opens the Build Options dialog as shown in **Figure 6.3**.



**Figure 6.3.** Building the project will normally cause each file that has been modified since the last time the project was built to be recompiled.

Programs will be recompiled to create new .FXP files, and menus will go through a generation process like they always have, but forms no longer go through a similar generation process. Instead, the object code is stored in the form itself, and will be recompiled if the program code in the form has been modified since the last project build.

You can force every file to be recompiled by checking the Recompile All Files check box. Because it is possible for object code in forms, class libraries, and reports to occasionally get out of sync with the program code, and timestamps of files can be inaccurate if you are moving components between machines, I usually keep this option selected to be sure that I've got the very latest changes, and that all of the object code in every form, class library, and report is compiled again.

If the Project Manager can't find a file during the rebuild process, it will display the Locate File dialog. You can choose to remove the file from the project, attempt to find the file, ignore the warning, or cancel the process. If the Project Manager comes up with errors during the rebuild process (syntax, missing files, or the like), the message in the status bar at the end of the build process will report on the number of errors found. You can display an error report automatically by opening the Project menu, selecting Build Options and then selecting the Display Errors check box. (If you forget to check the Display Errors check box, you can view errors by opening the Project menu and selecting Errors.) The Build Options dialog box allows you to control how your application is compiled.

Oftentimes, I will rip through the build process and select Ignore for all of the files, in order to get an idea of the number of errors. I've found it's a lot faster to fix a number of errors all at once than to stop and fix each one individually.

You can automatically run the application after the build process, which you may or may not choose to do based on your personal preferences.

The Regenerate Component IDs check box is used when building .EXEs and .DLLs. Selecting this check box will install and register any automation servers that are part of the project, and will generate new Globally Unique Identifiers (GUIDs) for any classes marked OLE Public.

You can have the Project Manager maintain version information for the .EXE or .DLL you are creating, as shown in **Figure 6.4**.

You can choose to explicitly assign three levels of version numbers, or have the Project Manager automatically increment the number for you. You can also assign additional information for the file you are building.

## The Project Manager context menu

Right-clicking any object in the Project Manager will open the Project Manager context menu, as shown in **Figure 6.5**. Various menu options are available depending on the type of object clicked.

The Expand All menu option is enabled if you've selected a node that has items below it. Exclude will mark the object so that it won't be included in the .APP, .EXE, or .DLL to be built. Set Main allows you to define which object is the top-most file in the application.



Figure 6.4. Clicking the Version button opens the .EXE file's Version dialog.

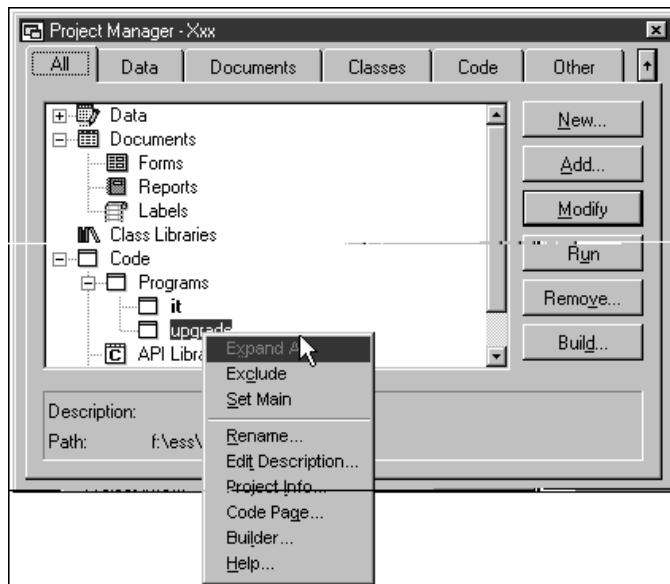


Figure 6.5. Right-clicking will open the Project Manager context menu.

Rename allows you to rename a file instead of having to go to the operating system to do so. Visual FoxPro is well-behaved with this functionality in that if you change the name of a menu or a form, for example, both the .MNX and .MNT (or .SCX and .SCT) file names are changed. Note that the generated files (such as the .MPR and .MPX) don't have to be renamed because they will be re-created when the project is built again. (However, the old versions of the generated files will still be lying around and you'll have to delete them manually.) Remember that when a file is called from another file, it's automatically added to the project. For example, if you call SCREEN1 from PROGRAM7, SCREEN1 will automatically be added to the project. If you change the name of SCREEN1 to SCRNI, you need to go back to PROGRAM7 and change the reference from SCREEN1 to SCRNI. When this happens to me, I just recompile all files in the project, and let the Project Manager list all of the unresolved references for me.

Edit Description allows you to add a text description to an object in the project. This is particularly handy if you use cryptic file names for the various forms, classes, and reports in your application—or as you add more and more files to a project, some of which begin to sound the same regardless of how explicit you were in naming them. To edit a file description, right-click the file to which you wish to add a description, and then click Edit Description. Depending on the letters used in the description (you can fit more letters like "I" and "l" than "M" and "W" in the same amount of space), you can use about 60 characters. This is purely descriptive and doesn't affect anything else about the project. The full path of the file is also shown next to the Path prompt at the bottom of the Project Manager.

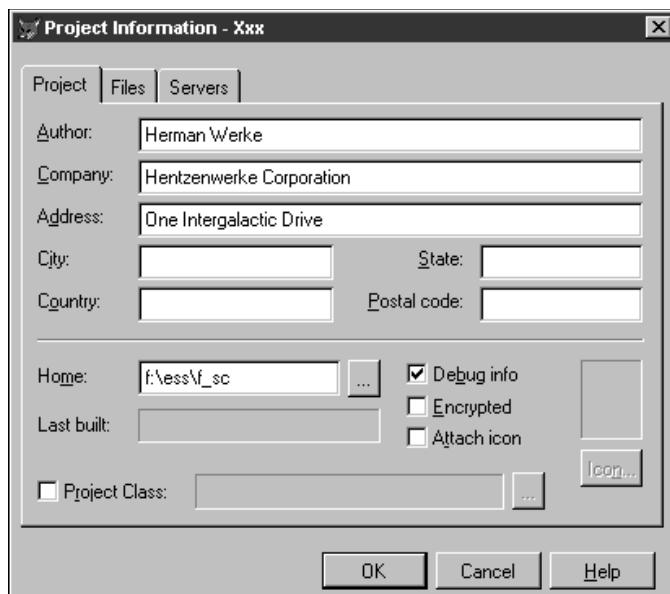
The Project Info menu command opens the Project Information dialog, as shown in **Figure 6.6**. You can include contact information, identify the location of the project file, and see the last date and time the project was built. You can include debugging information with the compiled file—if you select the Debug Info check box, you can view program execution in the Trace window. You can also choose to encrypt the resulting compiled file. Available third-party products purport to decrypt an encrypted application, but I have not tested any of these personally.

You can attach an icon to an application by checking the Attach Icon check box and then clicking the Icon button to select an icon. If the icon file you select includes both a 32x32 and a 16x16 version, they will be used as the icon attached to the file in Windows Explorer as well as the icon used if you create a Windows shortcut for the file. (You can assign a file-name value to \_Screen.icon in your application to change the icon in the title bar of your application.)

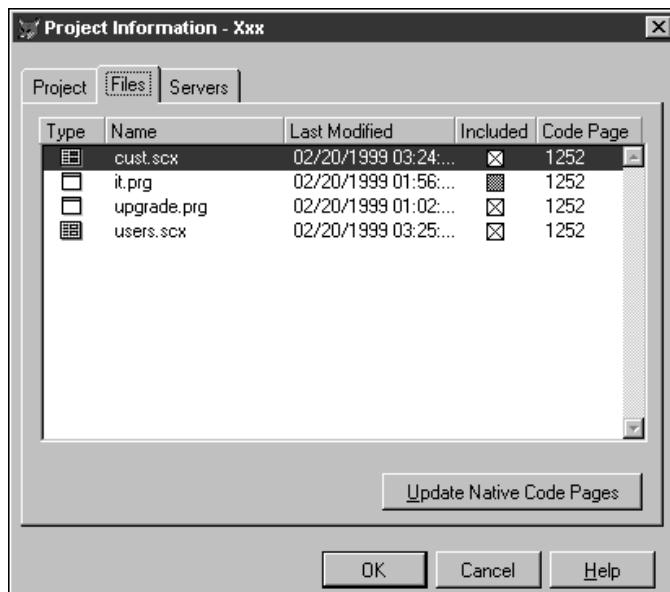
The Project Class area is used to specify a default ProjectHook class as a template for new projects. Project Hooks are covered in Chapter 18.

The Files tab, as shown in **Figure 6.7**, lists every file in the project in a list box control. This list box shows an icon identifying what type of file it is, the file name and extension, the date and time the file was last modified, whether or not the file is included in the project, and what code page is used for the file. The file with a filled-in check box in the Included column is the main program in the application.

Note that the Name column does not list the path—if you've got files from various subdirectories, you won't be able to tell from here, but the Path area at the bottom of the Project Manager window will tell you.



**Figure 6.6.** The Project tab of the Project Information dialog allows you to attach a variety of information to a project.



**Figure 6.7.** The Files tab of the Project Information dialog lists every file in the project.

The controls in the Included column are the only editable items in the list box—they operate just like standard check boxes.

You can resize the columns by positioning the mouse pointer on the separator bar between two column headings and dragging, and you can sort the list box on the values in a column by clicking the column header (you might have to hold the mouse button down for a second before releasing in order to get the column to sort). Holding down the Ctrl key while clicking will reverse the order of the sort.

The code page for .DBF files (other files use the .DBF format, such as .SCX files) is embedded into the file, but the code page for other types of files, such as text files, is not. You can use the Project Manager to keep track of the code page. For those files that don't have code pages, click the Update Native Code Pages button to update them with the current Windows code page.

The Servers tab will be discussed in the section on COM servers.

## The Project menu

The Project menu appears in the main menu whenever the Project Manager is the active window. Many of its options duplicate functions available through command buttons in the Project Manager window or the Project Manager context menu. New options include Errors, Refresh, and Clean Up Project.

When errors are found during a project build, they are placed in a file with the same name as the project but with an .ERR extension. You can view this file by opening the Project menu and selecting the Errors menu option.

The Clean Up Project menu option is used to get rid of old data in the project. Just like every other design surface in Visual FoxPro, the Project Manager stores its information in a table. The table has a .PJT extension, while the associated memo file has a .PJT extension. When you remove files from a project, they are tagged as deleted in the project table but not actually removed—just like deleting records in any other table. The project table also contains a full copy of the object code for each program and menu file. Object code for other files is stored in the files themselves as of version 6.0 of Visual FoxPro 6.0—that's why .PJT files are so much smaller than in the past. An active project might have dozens to hundreds of deleted records, and performance can suffer as the number of deleted records grows. You can use the Clean Up Project menu option to remove deleted records and winnow down the size of bloated memo fields. Note that the Clean Up Project menu option does not warn you or ask for confirmation first.

## Tips and tricks

I usually call the project and the main/calling program by the same name. I've found there are so many files around that using the same name for the project, program, and menu allows me to keep straight where the "starting gate" is. Otherwise, it's easy to forget the main name.

I call all of my projects by the same name, so the main program is also named the same. This way, when I'm bouncing around between various applications, I don't have to remember what clever acronym I used for a particular system.

If you have a file, such as a program, that is referenced indirectly, it will not be brought into the project automatically when you rebuild. For example, if your program specifically names a report in the code, like so:

```
report form MYREPORT to print
```

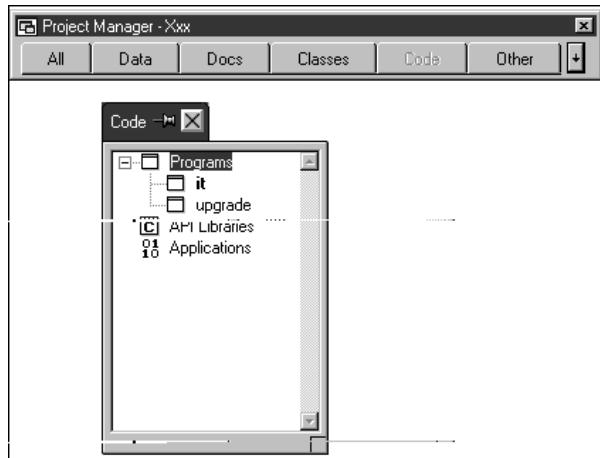
Visual FoxPro will pull MYREPORT.FRX into the project. However, you might have a situation where you pull the name of the report from a table based on the selection that the user made. In this case, you'd use indirect referencing to name the report, like so:

```
m.cReportName = "MYREPORT.FRX"  
report form (cReportName) to print
```

In this situation, Visual FoxPro will not bring “MYREPORT.FRX” into the project. The same technique works for other types of files. You can manually include them in the project (using the Add button) so that they are included in the final compiled file during builds.

As you dive head first into development, you'll find screen real estate at a premium. It makes sense to discuss some of Visual FoxPro's interface options that are applicable to the Project Manager so that you can keep it from taking over your desktop. First, remember that you can dock the Project Manager like any other toolbar, although you can only dock on the top or bottom of the screen. Double-clicking the half-height title bar also docks the Project Manager on the top of the screen. You can minimize the Project Manager anywhere on the screen by clicking the Minimize button in the upper-right corner of the window.

Suppose you often work with a specific type of file; you might want to have a list of those files open, but not the entire Project Manager window. You can “tear off” a single tab and keep it on the screen while the rest of the Project Manager stays minimized or docked. Click the box with the X to restore the tab to its rightful place in the Project Manager window. You can use the pushpin to keep the torn-off tab “on top” of all windows if you like. See **Figure 6.8**.



**Figure 6.8.** The tear-off tabs of the Project Manager allow you to work only with selected components of the application.

The Project Manager is a good repository not only for the files that make up the application, but also for the information related to the app. For example, you might have a to-do

list, memos to the customer, outlines for an instruction manual, and so on. You can include these as part of the project by adding them to the project under the Other, Other Files outline. The type of file defaults to a picture (.BMP) but you can change it to a .TXT file (File) or list all files by opening the List Files drop-down list. Then, just double-clicking one of those files in the Project Manager will bring up the application that was used to create that file. For example, if you add a Microsoft Word .DOC file to a project, double-clicking that file will open Word and load that file automatically. This works for Microsoft Excel and other applications as well. How can life get any better?



# Chapter 7

## Building Menus

**Well-designed Windows applications use a menu to provide access to the functions of the system. These menus take one of two forms: the new-style “toolbar” menu, used by 2000-era tools such as Office 2000, and the traditional menu bar/menu pad style, used by Visual Studio’s tools. Visual FoxPro’s Menu Designer allows you to create these traditional menus easily and quickly. While the Menu Designer and its resulting menu components are two of the only VFP components that still aren’t object-oriented, you can hook into them with third-party tools to provide additional functionality that accomplishes many of the same things. The Menu Designer also can be used to build shortcut, or context, menus.**

The string of words across the top of the application’s window, directly under the title bar, is the application’s menu. The entire line of words is called the menu, each word is called a menu pad, and each pad can launch a process, open a dialog, or expand into a drop-down menu. This drop-down menu can contain one or more menu bars (or “menu commands” or “menu options” according to which day of the week it is in Redmond), which themselves can either run programs or forms or expand into more drop-down menus.

Both menu pads and menu bars can be assigned hot keys, can be dimmed or completely removed to make them unavailable at selected times, and can have the default font changed.

### Driving an application with a menu

Much of my work over the last couple years has involved converting or replacing FoxPro 2.x applications, either in DOS or Windows, with newer, more robust systems written in VFP 6. I still find a remarkable number of applications that never used the Menu Builder in 2.x. Often, the developer instead used a mainframe, terminal-style, light-bar menu, where a list of choices was displayed in a single column in the center of the screen, and the user was required to press a letter or move a cursor key up and down through the column of choices. Other times, the main interface mimicked Access’s switchboard paradigm, with a series of command buttons launching into various functions. Don’t get me wrong; if the application works for the user, then I’m all for it. The only fault I find with these is that the developer didn’t take advantage of the available tools to make the application operate in as modern a fashion as possible.

In the fall of 1998, when Visual Studio 6 was released, the current state of the art in user interface dictated the use of a toolbar where each button actually contains a text label, and launches either a program or a form, or opens a drop-down menu. Visual Studio is a little behind the times, but not enough to be noticeable to the majority of users. The Menu Designer available to build menus doesn’t provide the toolbar mechanism, but the results can be very similar.

Current practice dictates a series of text prompts across the top of the screen, starting with “File,” continuing with “Edit,” and ending with “Tools,” “Window,” and “Help.” Other menu pads can be included in between these standard pads, as required by the application.

Database applications have somewhat different requirements than many other Windows applications, and I've adopted a couple of variations to this general practice.

First, the “File” menu often becomes “Forms,” and drops down into a series of menu options that open data-entry and query forms. With other software applications like word processors and spreadsheets, the user is quite cognizant of the file structure behind their work—each letter, memo, financial statement, or budget is represented on disk by a specific file. With databases, however, the file structures behind an order-entry form or a customer call record are usually not obvious: multiple tables, lookup files, and so on. Thus, it’s easier for the user if you map the menu options to forms that the user thinks of, and ignore the actual tables used behind the scenes.

I also usually insert one more pad, called “Operations” (or, sometimes, “Processes”) in between Edit and Tools. The drop-down menu for this pad contains options for various operations or processes that the user runs unattended, such as end-of-month postings, file conversions, version upgrades, and import and export processes.

While you’re certainly welcome to add more menu pads, you might want to restrict the number of menu pads and menu options underneath each pad so that the variety of choices doesn’t become overwhelming to the user. Office 2000 is attempting to do this by hiding little-used menu options, and displaying them once the user’s usage of the program has demonstrated that they use the menu command.

## Using the Menu Designer to create a menu

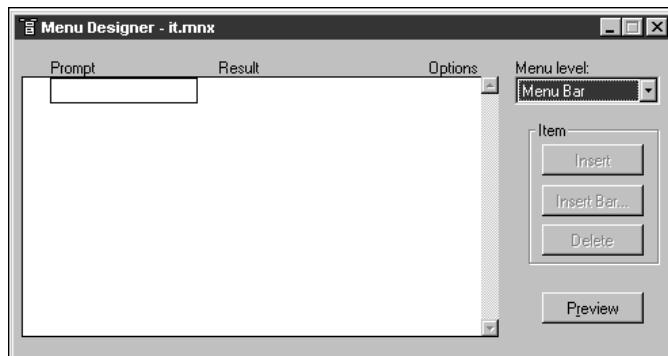
Follow these steps to create a menu:

1. Open the File menu and select New, Menu, New File. Then select New in the Project Manager, or type:

```
create menu XXX
```

where XXX is the name of the menu to create.

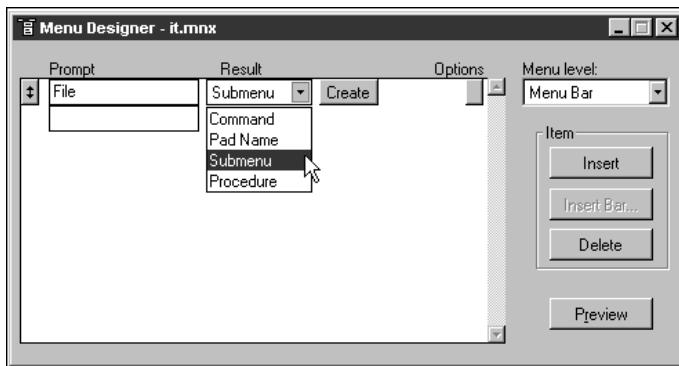
2. The Menu Designer dialog appears. See **Figure 7.1**.



**Figure 7.1.** The Menu Designer is a ‘data entry screen’ for creating Visual FoxPro menus.

This dialog is used to create the names of each pad and assign the action that executing that pad will perform. You can create drop-down menus by selecting the Submenu choice under the Result column in the Menu Designer. The Menu menu pad appears whenever the Menu Designer is the active window.

3. Type the text of the menu pad under the column labeled "Prompt." As you type, options for the menu will be displayed in the rest of the row.
4. Once you are done typing the menu pad text, either click or tab to the Result column. You can choose between Command, Pad Name or Bar #, Procedure, or Submenu. See **Figure 7.2**. Each of these choices is described in detail in the next section of this chapter.



**Figure 7.2.** The Result drop-down allows you to choose what action will be performed when the user selects the menu option being created.

5. Once you have selected a result (and, optionally, completed the required information for the result), click or tab to the Options column. Selecting the Option command button will display a Prompt Options dialog, from which you can control various aspects of the menu pad. These aspects include a keyboard shortcut, a Visual FoxPro condition that controls when the menu choice can be accessed, and the message that displays in the status bar when the pad is selected.
6. Click or tab to the next row of the menu and repeat steps 3-5 for each menu pad and drop-down menu.
7. At any time during the construction of your menu, you can examine what the final result will look like by clicking the Preview command button or opening the Menu menu and selecting Preview. The existing menu will be replaced by the menu you've built in the Menu Designer. You can make choices from the menu—commands are displayed in the Preview dialog, drop-down menus appear under their pads, and messages display in the status bar.

8. During the construction of your menu, and when you are finished designing it (and probably several times during its construction), you'll want to save it. Open the File menu and click Save to do so. You'll see the name of your menu displayed in the Menu Designer's title bar, with an extension of .MNX. Note that a corresponding file with an extension of .MNT is also produced; these files are discussed in more detail below.
9. Once you have completed the structure of your menu, you need to generate the actual menu program. (A behind-the-scenes look at what happens during the menu-generation process appears later in this chapter.) If you just want to create the menu, open the Menu menu and select Generate. (You will be prompted to save the menu if you haven't already.) When you're building projects, the menu-generation process comes along for the ride—you don't have to do it separately. The Generate Menu dialog appears and prompts you for an output file name. For the time being, accept the default menu name and click the Generate command button.

A thermometer bar will display and the menu program will be generated. Depending on the complexity of the menu you've constructed and the type of machine you are using, the generation process can take as little as a couple of seconds or as long as the better part of a minute.



*A menu program by itself isn't very interesting. Trying to execute it like any other program will produce unsatisfactory results—the current menu will be replaced by your menu, but executing a choice will not result in any action being performed. Furthermore, you will be stuck without any apparent method to return to the Visual FoxPro menu. See the “Tips and tricks” section later in this chapter.*

## Up close with the Menu Designer

### The Menu Designer dialog

The Menu Designer is where most of the action happens as you create and modify menus. It consists of three parts: the list box containing the menu options, the Menu Level drop-down box, and the Item command button group.

The list box enables you to add, edit, and reorder menu options. To add a new menu option, move the cursor to the blank text box in the Prompt column and begin typing a prompt.

The square command button to the left of the Prompt column (it contains a double-headed arrow when it has focus) is the mover bar for the main menu or submenu displayed. To change the order in which menu options appear, drag the mover bar (the rest of the row will follow) to the desired position. The row will displace the existing row and all rows below it will be moved down one row.

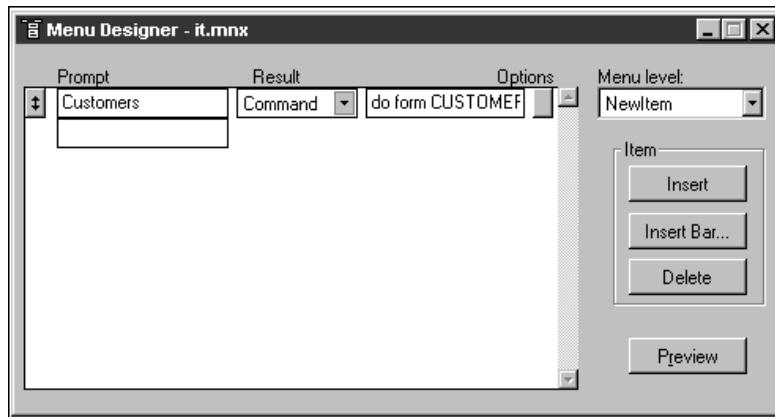
If you put a bar on the wrong pad by mistake, you're in for a bit of bad luck—there's no “move pad” function. You'll have to delete it from the errant pad and re-create it under the pad where you actually want it.

The Prompt column enables you to create menu text, assign access keys, and create dividing lines between groups of menu bars in drop-down menus.

The text typed into the Prompt text box will display as the menu text. The underlined letter in many menu options is called the access key. Holding down the Alt key and then pressing the underlined letter executes a menu pad access key. A menu bar access key is executed by opening the drop-down menu and then pressing the underlined letter. You can create an access key by preceding the letter of the menu text you want to use as the access key with a \< character combination. If you do not specify an access key, the first letter of the menu name will be used. A dividing line is the horizontal bar that separates two menu bars in a drop-down menu and is used to visually group sets of menu options for easier reference by the user. You can create a dividing line by typing \- in the Prompt column of an otherwise blank menu option.

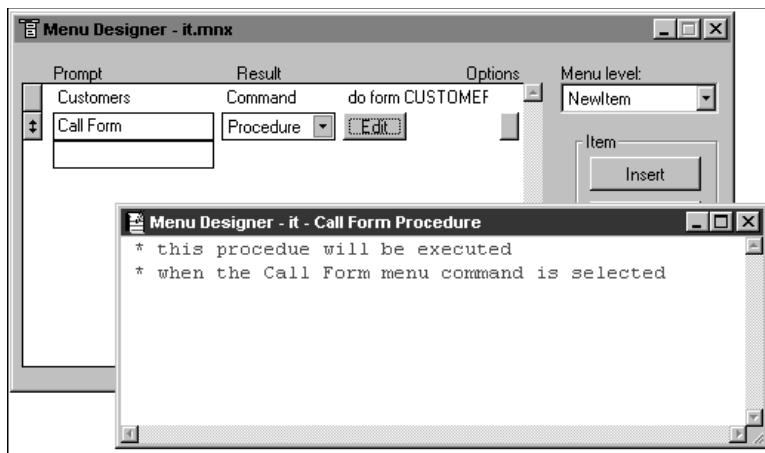
The Result drop-down list box (see Figure 7.2 again) allows you to select the action that will occur when the menu option is executed by the user.

Use the Command option when a single Visual FoxPro command should be executed as a result of the user selecting the pad, as shown in **Figure 7.3**. Enter the command to be executed in the text box between the Result and Options columns. You can also issue a call to a function with the Command result.



**Figure 7.3.** You can enter a command or a function call as a result of a menu option.

Use the Procedure option when more than one Visual FoxPro command should be executed as a result of the user selecting the pad. After selecting the Procedure Result, a Create command button appears in the next column. Clicking it will open a text-editing window in which the commands can be entered, as shown in **Figure 7.4**. If you're going to use the same procedure (a set of commands) in more than one menu option, you should consider using a command to call the procedure. Doing so will facilitate changes—you can make changes to only one procedure instead of searching for each menu option that uses the same set of commands. Once a procedure is created, the command button text changes to Edit.



**Figure 7.4.** The Call Form Procedure window is used to enter a series of commands that will execute when a menu option is selected.

Use the Submenu option when a drop-down menu should be displayed as a result of the user selecting the pad. After selecting the Submenu Result, a Create command button appears in the next column. Clicking it will change the appearance of the menu dialog—a new set of rows for the menu bars in the drop-down menu will be displayed—and the name of the menu pad that deploys this drop-down menu will appear in the Menu Level drop-down. Create submenus in the same way you created the main menu. Once a submenu is created, the command button text changes to Edit.

Technically, you can have a menu pad immediately launch a function, but that style is very much out of style. As a result, the Result column for all menu pads should contain Submenus.

The fourth choice in the Result drop-down varies between Pad Name and Bar #, depending on whether you are creating a main menu or a submenu. Both are used to identify the menu pad or bar that is created by the menu-generation process. If you do not provide a pad name or bar number, Visual FoxPro will create one for you. You would use Pad Name or Bar # if you needed to reference a pad name or bar number in a program. You should not reference the pad names and bar numbers that Visual FoxPro creates because they change during each generation. Because you can also assign a Pad Name or Bar # to a menu option by using a Command, Procedure, or Submenu Result type (through the Options command button), this is rarely used.

The Menu Level drop-down allows you to move between the main menu list box and the various submenu list boxes.

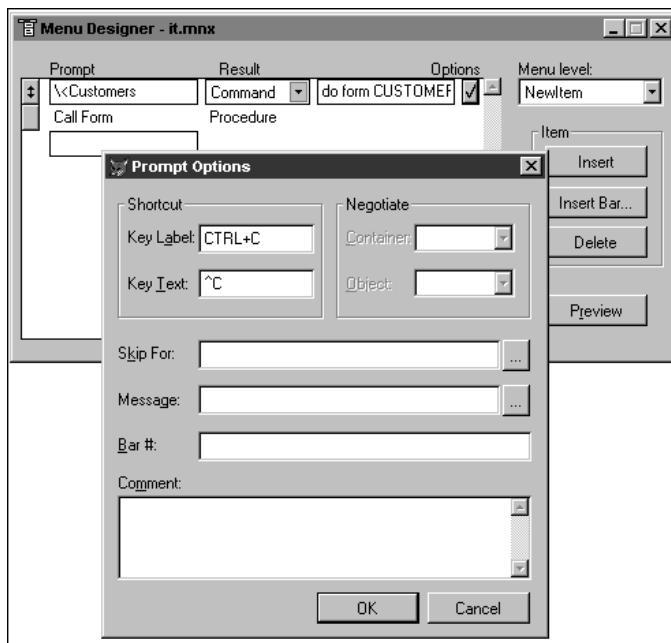
The Item command button group allows you to insert a menu option at the location of the cursor or delete the current menu option. Note that you will not be asked to confirm the deletion of a menu option when deleting it.

The Options command button allows you to assign keyboard shortcuts to menu options, create conditions to dim and enable menu options, display menu-option-specific messages in the status bar, and assign pad names for the menu option. The Options command button in the Menu Designer will contain a checkmark if any of the options are filled in.

A keyboard shortcut (also known as a hotkey) is the key combination displayed in some menu bars next to the menu name, as shown in **Figure 7.5**. It is executed by pressing the designated key combination. The difference between an access key and a keyboard shortcut is that the keyboard shortcut can be executed without having the drop-down menu open. You can create a keyboard shortcut through the Options dialog's Shortcut check box as shown in **Figure 7.6**.



**Figure 7.5.** The results of assigning a keyboard shortcut to a menu option.



**Figure 7.6.** The Shortcut area in the Prompt Options dialog box enables you to assign keyboard shortcuts to heavily used menu options.

The Key Label is the key combination that will execute the keyboard shortcut, while the Key Text is the actual word or phrase that will appear next to the menu bar in the drop-down menu. A typical use for Key Text is to abbreviate the key combination (for example, '^R' instead

of CTRL+R). Note that you can assign a keyboard shortcut to a menu pad but the Key Text will not be visible.

The Negotiate options allow you to define the location of user-defined menu titles in your application when a user edits an OLE object visually. These options are available only for menu pads.

A Skip For condition is used to control the access to a menu option. For example, in the Visual FoxPro menu, all of the menu bars in the Edit drop-down menu are dimmed if the cursor is not in a text-editing region. Once you click in a text-editing region and highlight a portion of text, the Cut and Copy menu options are enabled. If there is text on the text clipboard, the Paste menu bar is also enabled; otherwise it is dimmed. You might choose to provide visual clues to the users of your application by dimming and enabling menu options according to whether or not they should be available at the current time. You can use a Visual FoxPro expression in the condition to be evaluated as the program runs, so you can turn menu options on and off as desired.

You can specify a message to be displayed in the status bar when the menu option is highlighted. The button beside the Message text box brings up the Expression Builder, through which you can type a text phrase or enter an expression. If you use an expression, it will be evaluated during runtime and must evaluate to a character expression. This capability provides the ability to dynamically change the message according to various conditions—for example, you might use different messages according to the skill level of the user.

Finally, you can attach a comment to a prompt. This comment doesn't show anywhere in the user interface—it's merely a place for you to scribble a couple of notes if you've done something unusual, or if you want to document the functionality for the developer who's going to take over the maintenance of this application from you.

## **The Menu menu**

The Menu menu pad appears in the Visual FoxPro menu when the Menu Designer is the active window. This menu contains options to create a Quick Menu, to insert and delete options in the Menu Designer list box, to generate the menu program, and to preview the menu. The Insert, Delete, and Preview options are identical to the corresponding command buttons in the Menu Designer. The Generate function was briefly described earlier and is described in depth in the next section.

The Quick Menu option allows you to create a menu fully populated with the same menu pads and bars as in the standard Visual FoxPro menu. This is a useful tool for creating menus that can be examined to determine how Visual FoxPro's menus are put together as well as to create menus that contain menu pads and menu options that call native VFP functions, such as Edit and Help commands. The Quick Menu option is not available once you've created a menu that contains at least one pad.

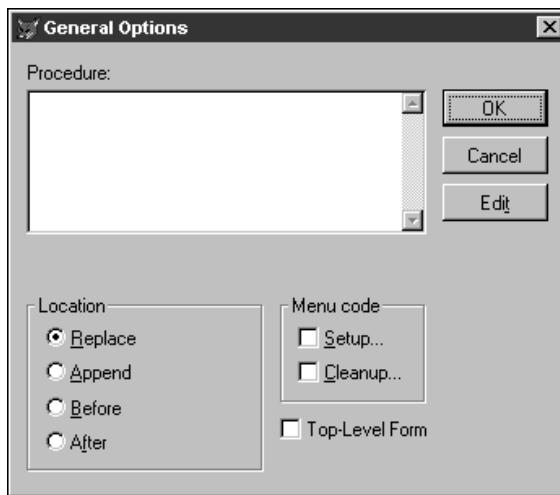
## The View menu

The View drop-down menu contains two additional menu bars when the Menu Designer is the active window. The General Options menu bar allows you to specify positions of the menu you are creating, and to create code that will execute before and after the menu is run, while the Menu Options menu bar allows you to attach a procedure to a specific drop-down menu.

### General Options dialog

Many applications require the ability to be modified as the program is being executed. Menu options can be enabled or disabled depending on what function the user is executing, or they can even be added afresh or completely removed from the menu. The General Options dialog box (see **Figure 7.7**) allows you to write code that will make those modifications based on conditions you specify.

The Menu Designer is actually just a “front end” for creating a menu program. As with any program, you can provide multiple paths for the program to follow depending on conditions your program evaluates or choices that the user makes.



**Figure 7.7.** The General Options dialog is accessed from the View menu when the Menu Designer window is active.

Setup code runs before the menu itself runs. A typical menu (take a look at the generated .MPR below to follow along more closely) consists of one or more DEFINE PAD commands that create the menu pads, DEFINE POPUP commands that create the drop-down menu, and DEFINE BAR commands that create the bars for the various drop-down menus. In addition, the menu program contains a number of ON SELECTION PAD and ON SELECTION BAR commands that execute actions when a menu pad or bar is selected. Setup code often contains initialization variables that are used in the various DEFINE and ON SELECTION commands. For instance, if you’re using an expression in a message, the expression might be created or modified in the menu’s Setup code.

Cleanup code runs after all menu commands are executed. Typically, Cleanup code contains code that needs to run after the menu is created as well as procedures executed from more than one menu option.

In addition to Setup and Cleanup code, you can create a procedure that runs directly from the menu program.

```
*      ****
*      *
*      * 99.09.09           MENU1.MPR          21:26:53
*      *
*      ****
*      *
*      * Whil Hentzen
*      *
*      * Copyright (C) 1999 Hentzenwerke Corporation
*      * Milwaukee, WI 53217
*      *
*      * Description:
*      * This PROGRAM was automatically generated BY GENMENU.
*      *
*      ****

*      ****
*      *
*      *                         Setup Code
*      *
*      ****

wait wind "We are in the SETUP code snippet"

*      ****
*      *
*      *                         Menu Definition
*      *
*      ****

SET SYSMENU TO

SET SYSMENU AUTOMATIC

DEFINE PAD _r0219yeqc OF _MSYSMENU PROMPT "My 1st menu option" COLOR SCHEME 3 ;
  KEY ALT+M, ""
DEFINE PAD _r0219yeqs OF _MSYSMENU PROMPT "My 2nd menu option" COLOR SCHEME 3 ;
  KEY ALT+M, ""
ON PAD _r0219yeqc OF _MSYSMENU ACTIVATE POPUP mylstmenuo

DEFINE POPUP mylstmenuo MARGIN RELATIVE SHADOW COLOR SCHEME 4
DEFINE BAR 1 OF mylstmenuo PROMPT "menu 1"
DEFINE BAR 2 OF mylstmenuo PROMPT "menu 2"
ON SELECTION BAR 1 OF mylstmenuo wait wind "1"
ON SELECTION BAR 2 OF mylstmenuo wait wind "2"

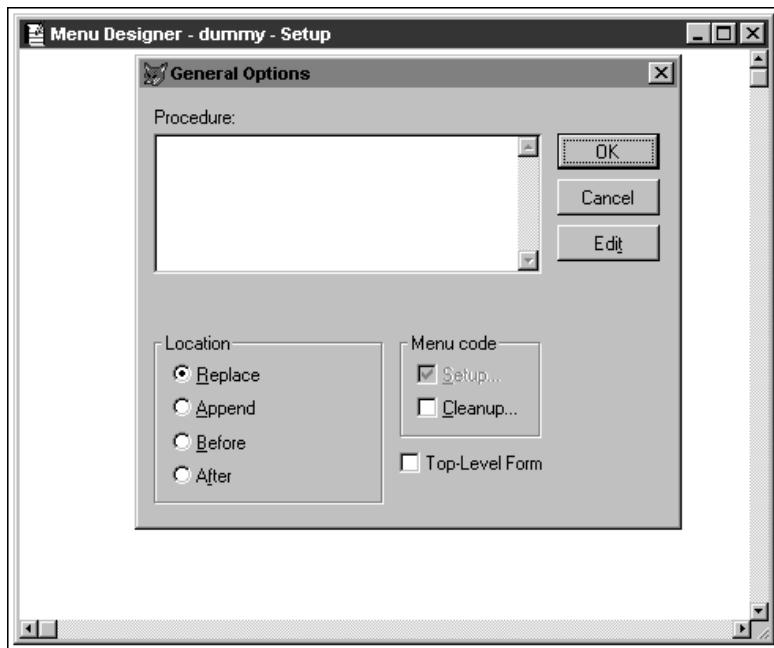
ON SELECTION MENU _MSYSMENU wait wind "We are in the General Options Procedure snippet"

*      ****
*      *
*      *                         Cleanup Code & Procedures
*      *
*      ****
```

```
wait wind "We are in the CLEANUP code snippet"
```

Accessing the Setup, Cleanup, and Procedure code from the General Options dialog is a little confusing at first because the interface is sort of stupid. When you bring up the General Options dialog (remember, the Menu Designer has to be the active window), the text-editing region is for the menu-level procedure. If the editing region isn't large enough, you can click the Edit button to open a full screen-editing window. However, doing so will not enable you to access the editing window—you need to click the OK button first to close the General Options dialog. This interface functionality has confused FoxPro programmers since the release of FoxPro 2.0, and you need to use the tool a few times to get used to it.

To access the Setup and Cleanup code snippets of the menu, select the appropriate check box in the General Options dialog. See **Figure 7.8**. Similarly, a text-editing window is brought forward, but you need to click OK to close the General Options dialog.



**Figure 7.8.** The Setup and Cleanup code windows are accessed by selecting the respective check boxes in the General Options dialog box, but you have to click OK in the General Options dialog before you can access the code window itself.

Note that while the text-editing window for one snippet is open, you can bring forward the General Options dialog and open the text-editing window for another snippet. But you'll need to close the General Options dialog in order to access the text-editing window, of course.

The Location option group in the General Options dialog allows you to determine how and where the menu is created. The Replace option will remove the existing menu and replace it

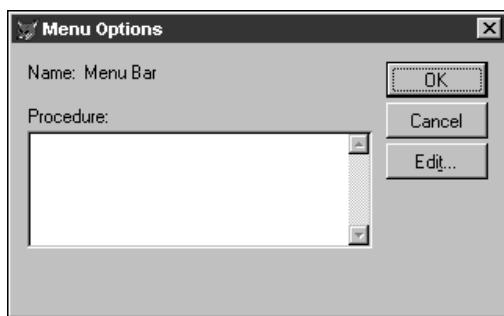
with the menu being created. The Append option will add the menu being created to the end of the existing menu. The Before and After options allow you to place the menu being created at a specific location relative to the existing menu. The pop-up menu that appears when selecting Before or After will show the current menu pads so that you can specify where the new menu pads will be placed.

And that Top-Level Form check box ... well, here's what this is for. Most Visual FoxPro applications use a desktop (or "parent" form) that hosts the rest of the application—including all of the forms in the app. You can create "top-level forms" in Visual FoxPro that act as SDI (single document interface) applications. These forms work at the same level as other Windows applications—at the same level as a regular VFP application with a desktop—and appear in the Windows taskbar.

You might want your top-level form to have a menu—but because the top-level form doesn't have a desktop that would hold the menu, the menu has to be attached to the form itself. This check box is used to create a menu that can be attached to such a top-level form.

### **Menu Options dialog**

Attaching a procedure to a specific drop-down menu uses the same type of interface as in the General Options dialog. First, use the Menu Designer to select the drop-down menu to which you want to attach code. Then open the View menu and select Menu Options to open the Menu Options dialog, as shown in **Figure 7.9**. If the procedure code is short, you can type it directly in the text-editing box. Otherwise, click the Edit button to open a text-editing window. Again, you must click OK to close the Menu Options dialog.



**Figure 7.9.** The Menu Options dialog can be used to add a procedure to a specific menu bar.

## **The Menu-generation process**

When you use the Menu Designer to create a menu, you are actually placing records that contain information about the specific menu options into a table. As a result, you can think of the Menu Designer as a "data entry screen" that places specialized records into a table for you. The table has an extension of .MNX, and its related memo file (information such as commands and procedures is actually stored in memo fields) has an extension of .MNT. When you are done with the Menu Designer and want to generate the menu program, use the Menu, Generate

menu option. Selecting Generate runs a Visual FoxPro program called GENMENU. This program opens the .MNX table and processes the data in the various fields, finally creating a text file with an .MPR extension. This .MPR file is just a Visual FoxPro program (a .PRG-type file), albeit with a different extension. To run the menu, issue this command:

```
do XXX.MPR
```

Visual FoxPro compiles the .MPR file into an .MPX file, just as a .PRG is compiled into an .FXP. Note that you must include the .MPR extension or Visual FoxPro will by default look for a .PRG file.

You can set a number of options when generating a menu. After selecting Generate, you'll be presented with the Generate dialog. Clicking the Options button will open the Generate Options dialog. You can choose where the generated code should be stored—with the menu's .MNX/.MNT files, with the project, or in another location.

## Tips and tricks

To remove a keyboard shortcut from a menu option, press the Enter or Backspace key. Pressing the Delete key will result in DEL being entered as the Key Text and Key Label.

Never modify the generated .MPR file. The .MPR is re-created each time you generate a menu, so if you make changes to the .MPR file, they will be lost when you regenerate. Alternatively, you would lose the capability to continue using the Menu Designer and would have to make all future changes to the .MPR, which subverts the intent of the Menu Designer in the first place.

It's confusing to understand when each of the code snippets—Setup code, Cleanup code, general procedures, and procedures attached to specific menu bars—is executed. Try placing a simple command or function call in each location generating the menu. Then examine the .MPR to see where each piece of code is placed, and run the menu to see when each piece of code is executed. I typically use a "wait window 'This is my setup code'" type of command that allows me to pause at each step during the execution of the program.

A tip my technical editor suggested was to minimize the windows of each of your procedures, and leave them open (but minimized) when you close the menu. When you next open your menu, all of the procedure windows will be opened the way you left them, so it's easy to spin through the windows to see where you placed code.

Many applications use some components of the standard Visual FoxPro menu, including pads for File, Edit, and Help, and at least a few of the menu bars under these pads, such as File, Exit; Edit, Cut; Edit, Copy; and Edit, Paste. As a result, the fastest way to create a new menu is to make a copy of an existing one and then delete undesired options instead of trying to create one from scratch. Additionally, if your applications have a lot of common functionality, you could create a "base menu" from which you can add selected options as necessary. Oftentimes, you'll use the same functions in some of the menu items across multiple applications. For example, many applications have a Tools menu pad under which functions such as System Preferences, User Maintenance, and Database Maintenance are found. Not only could the menu options be included, but the code or function calls for these functions should be part of this base menu.

Early in the creation of an application, you usually know what menu pads and options you're going to use. You can create them and then provide a "dead end" command so that when

you (or the user) select one, the system doesn't crash. The following command can be cut and pasted into every menu option and then replaced with the real command when available.

```
wait wind "The " + prompt() + " option is under construction. Hit any key..."
```

Note that the PROMPT() function automatically interprets which menu bar has been selected and includes its text as part of the Wait Window prompt.

Good menu design includes several components. First, it's accepted practice to use a set of three dots (called an ellipsis) at the end of a menu bar's name to indicate that the menu option will open a dialog. Furthermore, the use of an exclamation mark following the menu pad's name indicates that there is not a drop-down menu associated with this menu pad, although this style has gone the way of leisure suits and hot pants. These visual clues help users to figure out "what will happen when they select this" instead of making them guess.

Notice that Visual FoxPro automatically places an arrow at the end of a menu bar to indicate that another drop-down menu will be displayed upon the selection of a menu option with an attached submenu.

In complex systems, you might want or need to use a series of main menus. The main application would have one menu system, and the selection of any menu pad would then remove that main menu and replace it with another menu system, complete with multiple menu pads and associated drop-down menus.

You can use the PUSH MENU and POP MENU commands to place the active menu in memory and then run another menu temporarily. PUSH MENU places the active menu onto a "stack" in memory, and POP MENU pulls the most recently pushed menu out of memory and makes it active again. Note that the number of menus that you can PUSH depends on the amount of available memory.

When you're working with menus, it's easy to lock up the system by running a menu that doesn't contain a mechanism to return to the original menu. For example, suppose you're in the middle of creating a menu and you run it. However, the menu doesn't have an Exit option, and suddenly you have no way of returning to the Visual FoxPro menu or executing functions—except for exiting FoxPro and loading it again. Typing the command:

```
set sysmenu to default
```

will restore your menu to the default menu configuration. It's important to note that while the standard Visual FoxPro menu (File, Edit, View, Format, etc.) is originally the default menu, you can change the default with this command:

```
set sysmenu save
```

Doing so will prevent you from returning to Visual FoxPro's standard menu without exiting, so be careful when defining menus this way. You can return to the standard System Menu via SET SYSMENU NOSAVE followed by SET SYSMENU TO DEFAULT.

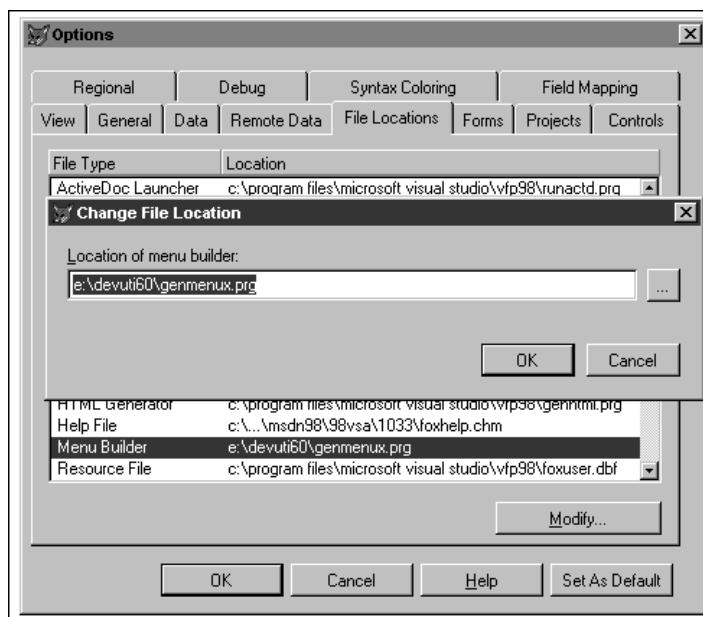
The Visual FoxPro menu is mostly a carryover from earlier versions of FoxPro, but it is still a powerful tool; with some tinkering, you'll find that you can usually bend it to your will.

## Adding flexibility to the Visual FoxPro Menu Designer

While the Visual FoxPro Menu Designer, despite its age, provides you with a fair amount of flexibility, it has its limitations. A public domain utility, GENMENUX, written by Andrew Ross MacNeill, provides additional flexibility to the capabilities of the Menu Designer:

- Control default menu positioning, colors, and actions without manually changing the .MPR file.
- Remove menu pads and bars based on logical conditions instead of simply disabling them using SKIP FOR.
- Automatically add hotkeys to menu pads.
- Call menu “drivers” at various points in the menu-generation process.
- Define menu templates that contain standard menu objects that can be inserted at any time into an existing menu.

GENMENUX is not a replacement for GENMENU. Rather, it is a Visual FoxPro program that acts as a “pre-processor” and a “post-processor” for Visual FoxPro’s original GENMENU program. To use it, place GENMENUX.PRG in a directory where Visual FoxPro can find it, or where you point to it in the File Locations tab of the Tools, Options dialog. Some developers place GENMENUX.PRG in the root directory of VFP; I put it in a directory of developer utilities, as shown in **Figure 7.10**.



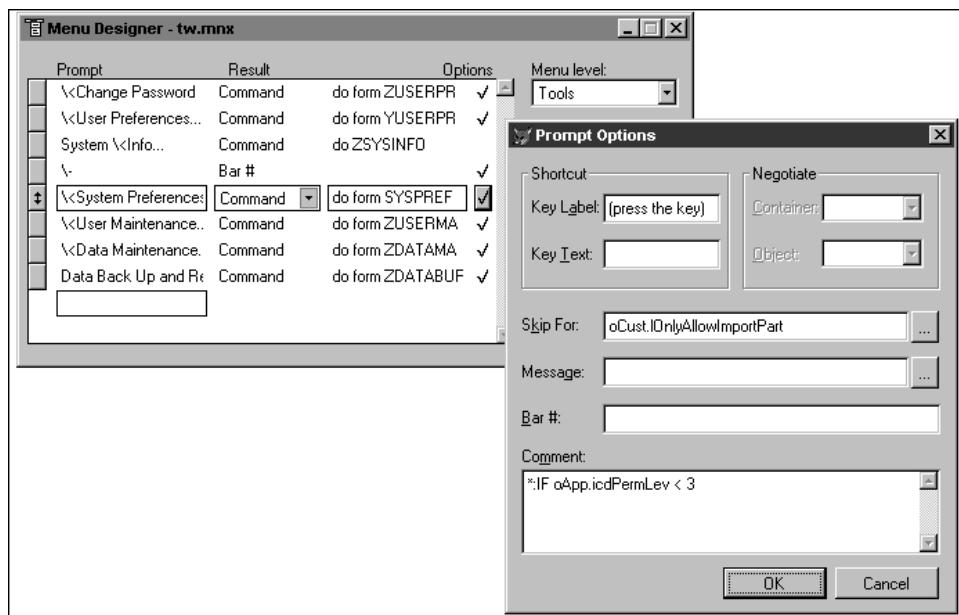
**Figure 7.10.** I place GENMENUX.PRG in a developer utilities directory and point VFP to it through Tools, Options.

You can also assign the location of GENMENUX to the \_GENMENU system variable. I discuss these types of variables in Chapter 14. After pointing VFP to GENMENUX, your next step is to add GENMENUX directives to the menu through the Menu Designer. These directives take the form of comments like so:

```
*:IF upper(oApp.cUserName) = "HERMAN"
```

The asterisk/colon combination is an indicator to GENMENUX that this comment is a GENMENUX directive and not just a silly string of characters that someone mistakenly typed into the Comment field of the Prompt Options dialog. See **Figure 7.11**.

When Visual FoxPro then generates a menu, GENMENUX will be called instead of GENMENU. It makes a copy of the .MNX menu table file and manipulates the copy. This manipulation might involve changing the order of some records, changing the contents of other records, and otherwise altering the file in ways that you can't do with the Visual FoxPro Menu Designer. When finished with its manipulations, GENMENUX then calls GENMENU, which does the usual menu generation, albeit with the modified copy.



**Figure 7.11.** A GENMENUX directive is placed in the Comment field of the Menu Designer's Prompt Options dialog.

How does GENMENUX know what manipulations to make, and when? The Comment field in the .MNX file contains the GENMENUX directives that act as flags that tell GENMENUX to do some special processing. The \*:IF directive is a flag that indicates that this menu option should be removed (not disabled, but completely removed) from the menu if the

condition following the IF statement is false. GENMENUX actually modifies the .MPR file. In this case, the following code was added to the .MPR file:

```
IF NOT (upper(m.gcUserName) = "HERMAN")
    RELEASE BAR 6 OF Tools
ENDIF
```

When this program—the menu is just a program, right?—executes, the contents of the global memory variable, m.gcUserName, are compared to the string “HERMAN” and if they’re not equal, the sixth menu bar of the Tools menu is removed.

This capability, obviously, is quite powerful: the ability to modify the menu by interpreting the menu program on the fly as it’s executed.

 Space doesn’t permit a listing of all the GENMENUX directives—or even a fuller explanation of all of the things you can do with it. But a copy of GENMENUX, together with documentation of more than 75 directives, is included with the source code downloads for this book at [www.hentzenwerke.com](http://www.hentzenwerke.com).

## Creating context menus

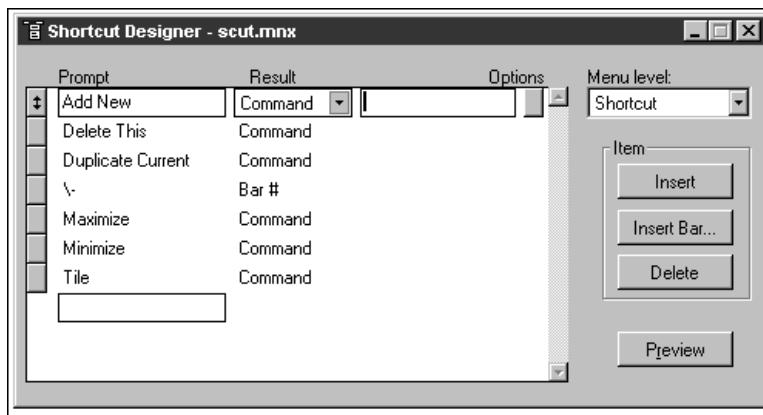
A context menu (also known as a shortcut menu) is the drop-down menu you see when you right-click in various areas of the IDE. You can add context menus to your own application.

To create a context menu, issue the CREATE MENU command and click the Shortcut menu button that displays in the resulting dialog, as shown in **Figure 7.12**.



**Figure 7.12.** Click the Shortcut menu button in the New Menu dialog to create a context menu.

The Shortcut Designer will appear as shown in **Figure 7.13**. It works just like the Menu Designer except that you don’t create menu pads across the top of the window. Create the menu options just as you did when you worked with the Menu Designer. You can even create drop-down menus, although you should tread carefully because, by definition, a context menu should have just a few menu options that are relevant to the specific context it’s used in.

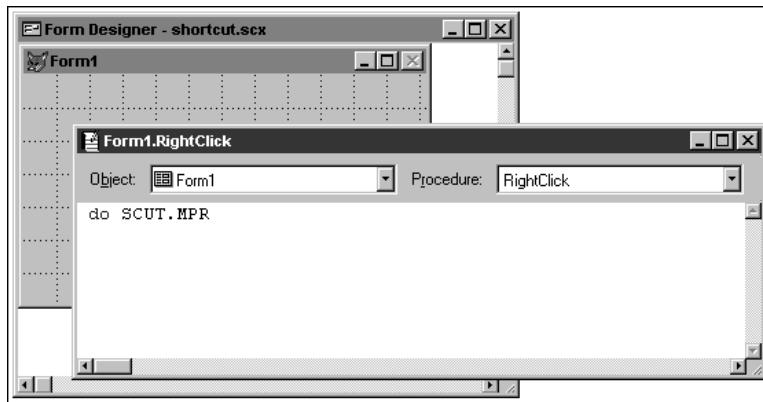


**Figure 7.13.** The Shortcut Designer works much like the Menu Designer.

Add commands, function calls, or procedures as you would in the Menu Designer. When you're done with the menu options, open the Menu menu and select Generate to create the context menu .MPR file.

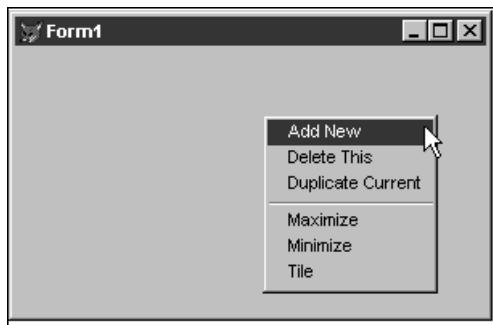
The final step is to attach the context menu to the location where it will be called. In this example, I've called the SCUT.MPR program from the right-click method of a form by entering the following code into the right-click method, as shown in **Figure 7.14**:

```
do SCUT.MPR
```



**Figure 7.14.** Call a context menu by attaching a call to it in the RightClick method of a form or control.

I could have just as easily attached the shortcut method to a specific control on the form, of course. Finally, running the form and right-clicking anywhere on the form opens the context menu, as shown in **Figure 7.15**.



**Figure 7.15.** Right-clicking on the form opens its context menu.

Before I go on, I want to mention one thing that you'll want to come back to after you've gone through Chapter 13. Unlike regular menus, you will likely be calling methods of the object that invoked the shortcut menu. In other words, you won't be running stand-alone programs from the "Add New" or "Duplicate Current" menu bars in this shortcut menu. Instead, you'll be calling methods that belong to Form1.

The methods belong in the Form (actually, the class that the form was created from) because they're a behavior you want this type of form to have. (As I said, this will make more sense after you've gone through Chapter 13 a couple of times.) As a result, you'll want to use commands for these menu options that look like:

```
oFormObject.AddNew()
```

where AddNew is a method in the form class that was used to create Form1. You'll need to pass a reference to the form, however, to the menu, so you'll actually use syntax like:

```
do SCUT.MPR with THIS
```

where "THIS" is referring to the form itself. (The command "do SCUT..." is in the form, right?) Yeah, a little ethereal right now, but like I said, come back to this.

If you're not comfortable with the forms and methods I've discussed here, you might want to revisit this section after you've covered Chapter 8, "Building Forms."

## So why isn't the Menu Designer OOP-ified?

One of the biggest complaints of VFP developers is that its menu designer isn't object oriented. One of the two big changes between FoxPro 2.x and VFP 3.0 was the introduction of object orientation into the Form Designer. The change was big enough that it was easy to forgive omissions elsewhere in the product. Surely they would be corrected in future releases. When 5.0 was released, the big news was new developer tools (and, some say, the lack of 16-bit and Macintosh versions). Version 6.0 was released and there still isn't an object-oriented menu. It's obvious that this particular mechanism is not in the plans.

Why not, and what should you do about it?

First of all, Visual FoxPro is a Microsoft tool, and, as such, is subject to Microsoft's vision of application development. They have stated incessantly over the past few years that application development is moving from monolithic, single-purpose applications to component-based systems that sport flexible and varied user interfaces. Visual FoxPro's role in this vision is that of a middle tier, and, as such, doesn't have a heavy-duty requirement for broad interface tools like menus. For those of you that still build LAN-based applications, the current menu building tools should be good enough for you to get by. At least that's the word coming from Redmond.

So what should you do about it? If you're bothered enough, you could create your own object-oriented menu tool, use one in a third-party commercial tool like Visual FoxExpress, or find a freeware or shareware version such as at Cornerstone Software's site ([www.cornerstone.co.nz](http://www.cornerstone.co.nz)). Myself? My users are more worried about the forms and reports in their applications, not whether or not they can customize the various buttons on the menu bar. I'd suggest you spend your time worrying about something important—like why the "operating system suitable for mission-critical applications all over the world" still GPFs if you look at it wrong.

# Chapter 8

## Building Forms

**The form is where the action is. Most applications need to interact with a user at some point in time, and the form is usually where they do it. In this chapter, I'll discuss the VFP Form Designer, how to create and run simple forms, and how to place and use basic controls. In subsequent chapters, I'll discuss each control in depth and then use both forms and controls to introduce Visual FoxPro's implementation of object-oriented programming. But first, let's build some forms.**

The Form Designer is Visual FoxPro's tool for creating an application's user interface, and is the replacement for the Screen Builder in FoxPro 2.x and earlier. Forms are used for data-entry screens, dialog boxes, and so on. As with the other tools in Visual FoxPro, you can take a number of routes to create forms, including the Form Wizard, Quick Form, and creating a form from scratch. In addition, Visual FoxPro's object-oriented paradigm greatly expands the scope of the Form Designer's capabilities and how you go about creating forms. Nowhere in the product is object orientation more evident than when you're creating forms.

However, in this chapter, I'm going to almost completely ignore object-oriented programming, instead just focusing on the Form Designer and its related tools. There are two reasons for doing this. First, forms are a lot to bite off, and I think that trying to learn the new tools as well as the concepts of object-oriented programming is too much to handle in one sitting. There are many windows, dialogs, toolbars, and menu options that you'll use during the form-building process, and you'll want some practice just to get comfortable with where everything is.

Second, the hallmark of object-oriented programming is planning—many in our industry estimate that the planning and design phase of an application that takes full advantage of object orientation should run nearly 70% of the total development time—a far cry from the 10 to 20 percent usually spent now. However, it's likely that you live in the real world, and that means that your management or customers are going to want to see you writing code, not doing “all that other stuff.” Too often we hear the line “I'll find out what the users need when I get back from my trip next week but you should start programming now.” As a result, you're going to have to deliver—and you can—now. It's not an optimal process, to be sure, but it's practical. While you're learning the tools, you can be planning on how to migrate to object orientation. Most experts (and anyone who has delivered a couple of applications) will tell you that you can expect to throw out your first few attempts at designing classes. By developing skills with the tools first, you'll be able to reduce the number of attempts that you throw out because you won't have to rework your classes to handle technical mistakes—just design issues.

### The big picture

Previous versions of FoxPro, as well as other “screen painting tools,” came with a set of controls, such as data elements, radio buttons, and push buttons that you placed on a screen. Once you did so, that control was there to stay. If you needed to modify it, you did so directly. If you placed the same type of control on a number of screens and then decided that some

aspect of that control needed to be changed, you had to change every copy of that control individually.

Visual FoxPro takes this mechanism and expands it in a number of different ways. First, it comes with an expanded set of controls that you can place on forms—including OLE objects, timers, and page frames (also known as tabbed controls). Next, instead of just placing controls on a form directly, you can create “copies” of the controls, customize those copies, and then place “copies” of the customized copies on your forms. For instance, if you wanted all your command buttons to be bright purple with an Ice Age font, you could make a custom command button with those properties and then use that as your “master” to create command buttons for each new form.

Yes, you could do this in earlier versions, but the capability wasn’t native to the product and required some workarounds. To introduce a bit of object-oriented terminology (I’ll go into more depth in the next chapter), this capability is called inheritance: the new controls inherit properties from the master. Visual FoxPro’s set of controls is the base class of controls, while the groups of customized controls you create as well as the copies of those customized controls are subclasses.

Furthermore, and this is the cool part, the copy of your customized object that you’ve placed on a form isn’t really an actual copy—it’s a reference to your customized object. (You can actually go into the form definition and, roughly, see that there is a reference to the file name of the master control.)



*This is significantly different from how Visual Basic works, where you can create a definition of an object and then use that definition to create “copies” of it, say, to place on forms. The difference is that the copy on the Visual Basic form has no link back to its definition—all of the information in the definition is copied directly to the copy. If you change the definition, nothing happens to the copy on the form. In other words, there’s no inheritance.*

The effect is that if you change the original customized object, each of the controls that reference that object automatically change! So if you decide that purple command buttons aren’t such a good idea after all, you can change the color of your master purple button to red, and all of the purple buttons on each form will automatically change to red. You won’t have to make the changes to each individual purple button. Again, the actual command button on the form inherits the color property from your custom class.



*Before I get too far past this brief example, I want to mention one thing. I’ve used an example of a purple button with one font and a red button with a different font to demonstrate the concepts behind VFP’s object-oriented behavior. However, in practice, you typically wouldn’t use object orientation to inherit physical attributes like size and color. Instead, your goal should be to inherit behaviors and operational mechanisms. More on this in Chapter 10.*

And this capability wasn’t available in earlier versions of FoxPro at all. The mind boggles at the power this provides, doesn’t it? And once the mind is done boggling, we realize that

we've got to learn two things: how to use the Form Designer with all of its functions and controls and features; and then how to design these custom controls and how the referencing capabilities work. The above explanation is a brief summary of object orientation in Visual FoxPro—the subject is far more complex than what I've alluded to here—but even by opening the door a crack, you can see that this would be too much for one chapter, so I'm going to break it into several pieces.

In this chapter, I'm going to address the Form Designer—how to use it to create forms, place controls on the forms, and how to write code to “make things happen.” But this chapter assumes that you're only going to copy objects from the set of master objects that VFP provides. And it's important to understand that this is NOT the way you're going to do development in “real life”—but then, you only used training wheels while you were learning to ride. Once you're comfortable with the tools used to build forms, we'll look at creating these customized objects and how to use them “for real.” I can't stress this enough: In this chapter, you'll become familiar with the interface of the Form Designer, learn where code is, how to run forms, what the different controls are, and so on. There's a lot to learn here, so I'm not going to clutter up your brain any more than necessary. Once you've mastered the Form Designer, we'll talk about using the object-oriented capabilities of Visual FoxPro.

## Terminology and tools

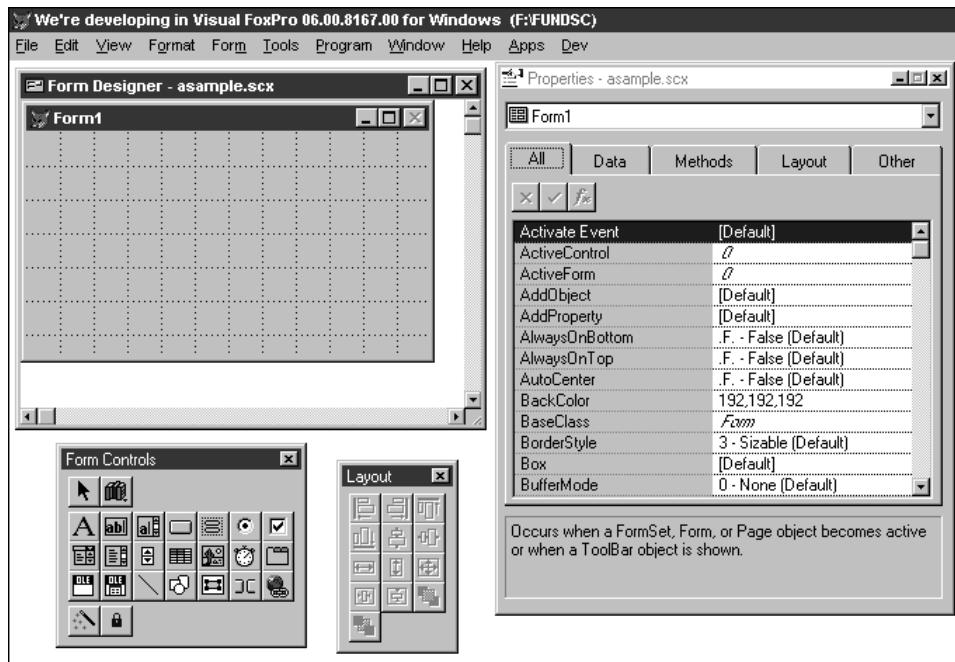
Forms can be simple affairs with a few buttons and perhaps a bit of text, or they can be hideously complex with several forms tied together in a “form set” and containing, collectively, dozens (or hundreds!) of controls that have any number of permutations and rules governing what happens in any given event. Before we create a form, let's go over the various pieces and tools that we're going to deal with.

The first thing to stress is that a control is any object—not just those that “do things” such as command buttons. Text boxes (where you type data), and edit boxes (where you type in data for memo fields) are also considered controls, as are option buttons, spinners, and so on.

Next, when designing forms, you're actually going to use several tools, as shown in **Figure 8.1**. The Form Designer is a visual design tool, much like the screen painting surfaces of other currently popular fourth-generation languages. The Form Controls toolbar provides a palette of controls, such as text boxes, command buttons, and check boxes, from which you can select controls for your form. The Properties window allows you to view and modify the attributes of the form and each control on the form. The Layout toolbar provides a number of shortcuts for manipulating objects on the form.

The form acts as a “container” that holds additional objects, or controls. Each control that you place on a form (or “in a form,” if you wish) has attributes that describe and define it. These include properties, such as Size and Color, and events, such as Click and GotFocus. Controls might share some attributes but not others—both a command button and a text label have a Click event, since you can click on both of them, but a command button has a Cancel property (which specifies whether the button is the Cancel button) and a text label does not. A control's properties have values, such as .T. for FontBold or 128,128,128 for ForeColor. Properties are grouped into three categories: Data, Layout, and a catchall category, Other. You can attach code to the events that belong to a control. This code would be executed when the event happens. This code is referred to as a method. A method is a set of commands to be

executed when an event is fired: “Do these commands when the user clicks on this control.” It’s simply a function or a procedure.



**Figure 8.1.** The Form Designer window, the Properties window, and the Form Controls and Layout toolbars are all parts of the environment used when creating forms.

Referring to forms, controls, and properties requires new syntax, and additional functions and keywords. In brief, you’ll refer to each of these objects using a naming convention that begins with the object closest to the top of the containership hierarchy and then moves down to each contained object. For instance, to refer to the foreground color property of a command button named cmdDone in a form named frmLookUp while inside a form, you’d use the following type of syntax:

```
frmLookUp.cmdDone.ForeColor
```

It’s often useful to be able to use generic code to refer to the form, and you can do so with the THISFORM keyword (which is similar to the ME keyword in other languages). Thus, you could use this alternative:

```
thisform.cmdDone.ForeColor
```

This would prevent you from having to know the name of the current form. And if you’re referencing whichever form happens to be active, from anywhere in the application, you could use the \_SCREEN.ACTIVEFORM keyword, so your command would look like this:

```
_screen.activeform.cmdDone.ForeColor
```

Running methods similarly requires new syntax as well as additional functions and keywords. For example, to call the Refresh method of a form from somewhere else in the form, your command would look like this:

```
thisform.refresh()
```

We'll delve into the hows and whys later; for now, follow along with each example to see how to reference various objects.

## Creating and running a form

The first form you're going to build will simply introduce you to the process of creating a form and running it. There are a number of significant differences between Visual FoxPro and earlier versions, and I'll cover many of them immediately. Follow these steps:

1. Create a blank form by opening the File menu and selecting New, Form, New File.  
Or type:

```
create form XXX
```

where XXX is the name of the new form. Doing so creates a pair of files named XXX.SCX and XXX.SCT. These are a .DBF and matching .FPT file that hold the information for the screen. Each record in the table represents an object in the form.

The Form Designer window will appear. You'll notice that the Form Designer contains a smaller window named Form1. Several other windows and toolbars may appear automatically.

2. Open the other tools so they're available during form design.
  - Open the Form Control and Layout toolbars if they didn't automatically appear. Use the View, Toolbars menu option.
  - Dock the toolbars to the top of the screen for the time being. You might decide to change their position later, but as you're getting used to having a multitude of windows on the screen, this will keep a couple of them out of the way.
  - Open the Properties window if it's not visible. You can do this by right-clicking on the new form and selecting Properties, or by selecting the View, Properties menu option (the Form Designer must be the active window).
  - You'll notice that the Object combo box in the top of the Properties window shows that the current object being displayed is the object Form1. The properties shown in the list box portion of the Properties window are all specific to the form. As you add controls to the form, you can use the Object combo box to switch between various objects associated with the form, thus being able to view and modify the properties that apply to each object.

- Open the Code window if it's not visible. You can do this by right-clicking on the new form and selecting Code, or by selecting the View, Code menu option (again, the Form Designer must be the active window). At this point, you now have all the tools you'll need for designing a form. And you'll now see why a 17-inch monitor at 800 x 600 or better resolution is highly recommended.
3. Modify the form itself.
- Select the Properties window and change the title for the form by selecting the Caption property and typing in a new caption, such as "My Hello World Form" in the Property entry box under the tabs. Press Enter to save the new value.
  - Change the color of the form's background by selecting BackColor and clicking the ellipsis button next to the Property entry box, and then selecting an appropriate color, such as hot pink. As soon as you click OK in the Color Palette window, the form will change to reflect the new color you've selected.
4. Add a label control to the form.
- Place a label in the form by selecting the Label icon in the Form Controls toolbar and then clicking in the form. A crosshair will appear as you are dragging, and the label will appear once you let go of the mouse button. Notice that the contents of the Properties window have changed so the current object is the label you just created.
  - Change the caption of the label by selecting the Caption property, as you did with the Form Caption. Again, press Enter to have the change take effect. You might also need to widen the label so that the entire caption appears. To do so, you can either change the label's AutoSize property to True, in which case the label will be resized to the needed width (but no wider), or you can drag one of the sizing handles (the eight black boxes on the corners and sides of the label).
5. Add a "Hello World!" command button to the form.
- Place a command button in the form by selecting the Command Button icon in the Form Controls toolbar and clicking and dragging in the form to size the button. Once you are done, the button will appear with the caption Command1, and the Properties window will change to show the properties of the command button.
  - Change the Caption of the command button to "Say Hello" in the same way you changed the form and label captions.
  - Create a method that will display a "Hello World" message when the Command1 (Say Hello) button is clicked. Select the Code window, select the Command1 button from the Object combo box, and then select the Click event from the Procedure combo box. Then, type the following code in the edit region of the code window:

```
messagebox("Hello World!", 48, "Just Testing...")
```

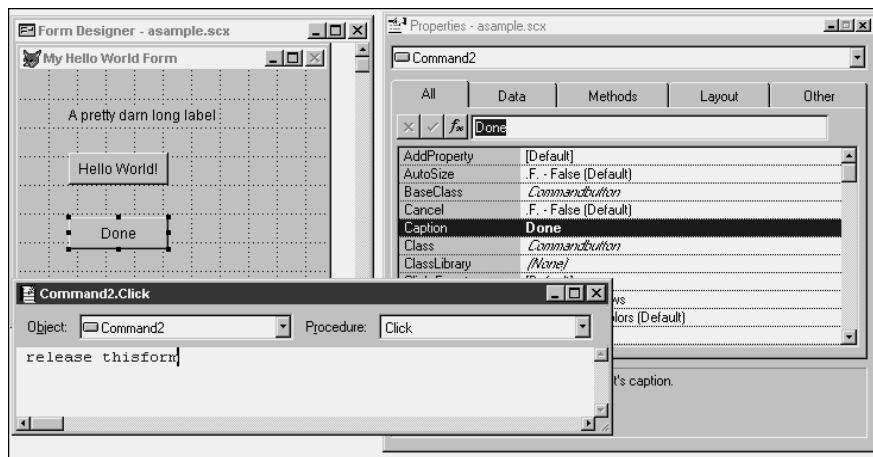
(For those of you who have used the MessageBox function in earlier versions of FoxPro, you no longer need to “set library to FOXTOOLS” before using MessageBox!)

6. Add a Done command button to the form.

- Place a second command button in the form. Note that this button will appear with the caption Command2, and the Properties window Object combo box now contains entries for the form, label, and both command buttons. Change the Caption to “Done.”
- Place a method that will terminate the form in the Click event of the Done command button by selecting the Code window, selecting Command2 from the Object combo box and typing the following code in the edit region of the code window (see **Figure 8.2**):

```
release thisform
```

(We’ll discuss the new language and syntax later.)



**Figure 8.2.** The Code window is used to attach code segments (methods) to specific events.

7. Run the form.

- You can run this form in one of about a thousand different ways. You can (1) click the Run (exclamation) icon in the standard toolbar, (2) right-click on the form and select the Run menu option, (3) select the Form, Run Form menu option, or (4) if you’ve already given the form a name and saved it, select the Program, Do menu option, and then select the form name from the dialog.
- You’ll be prompted to save (or resave) the form and the code for Command1 and Command2.

- The form will appear. You can click the Say Hello button, move the form around, resize it, and so on. When you’re finished playing with it, click the Done command button to get rid of it.

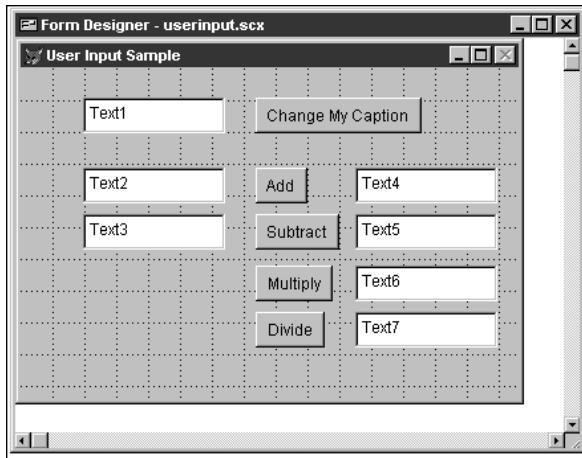
Users of earlier versions of FoxPro will notice that there is no intermediate “generate” process. In essence, Visual FoxPro runs the .SCX “as is” instead of running the .SCX data file through a generate program to create a .PRG-like file. The object code actually executed is stored in the .SCX along with the rest of the data elements.

## Changing form properties through user input

Now that you understand how to build and run a simple form and how to access the various controls and tools you need to use to do so, let’s create a form that accepts user input and does something with it. This form will take the value that you enter into a text box, and change the caption of a button to that string. Then it will allow you to add, subtract, multiply, and divide two numbers that you enter on the form, as shown in **Figure 8.3**.

1. Create a blank form.
  - Change the following properties:

**Caption: User Input Sample**  
**Name: userInput**
  - Save the form with the file name “UserInput.”
  - Resize the form by dragging the corners so that it takes up about a quarter of the screen. If you enlarge the Form Designer window, you’ll see a white rectangle that represents the maximum size of a form that can be designed. (The size of this rectangle is determined by an option in the Forms tab of the Tools, Options dialog.) Once you’ve got the form approximately sized, use the Height and Width properties to specify its exact size.
  - You can turn the Grid Lines on or off with the View, Grid Lines menu option; enable or disable the Snap To Grid functionality with the Format, Snap To Grid menu option; and enable or disable the position display in the right side of the Status Bar with the View, Show Position menu option. If you want Grid Lines on, you can control the spacing of the lines with the Format, Grid Scale menu option. The unit of measure used by each of these is controlled by the Scale Units combo box in the Forms tab of the Tools, Options dialog; by default, it’s Pixels.



**Figure 8.3.** The User Input Sample form.

2. Place seven text boxes and five command buttons on the form as shown in Figure 8.3. Set their properties as follows:

```
command1
AutoSize: .t. - True
Caption: Change My Caption
```

```
command2
AutoSize: .t. - True
Caption: Add
```

```
command3
AutoSize: .t. - True
Caption: Subtract
```

```
command4
AutoSize: .t. - True
Caption: Multiply
```

```
command5
AutoSize: .t. - True
Caption: Divide
```

3. Enter the following code into the Click event of each command button.  
(You can double-click the button and select the Click event from the Procedure drop-down menu.)

```
command1.click()
this.caption = thisform.text1.value
```

```
command2.click()
thisform.text4.value = thisform.text2.value + thisform.text3.value
```

```
command3.click()
thisform.text5.value = thisform.text2.value - thisform.text3.value
```

```
command4.click()
thisform.text6.value = thisform.text2.value * thisform.text3.value

command5.click()
thisform.text7.value = thisform.text2.value / thisform.text3.value
```

4. Enter the following code into the Init event of the form. (You can double-click on the form itself and select the Init event from the Procedure drop-down menu.)

```
thisform.text2.value = 0
thisform.text3.value = 0
```

5. Run the form.
  - Enter a text string into the text1 text box, and click the command button to its right. The caption of the command button will change to display the string.
  - Enter two numbers in the bottom two text boxes on the left (text2 and text3), and click the various command buttons to their right. You'll see the results display in the text boxes next to the command buttons. Note that you'll be prohibited from entering non-numeric values. Furthermore, if you enter a zero in the bottom text box and then try to divide, you'll get a value of "\*\*\*\*\*" in the corresponding text box.
  - The code entered in the Init event defines the contents of the two text boxes as numeric—if this code hadn't been included, Visual FoxPro would have interpreted an entry of "2" as a string instead of a number, and would have had the greatest difficulty in multiplying or dividing the two values.
6. Modify the form while it's running.

You'll notice that while the form is running, the Command window appears again. This means the form is running, but you can issue interactive commands as well. Try entering the following commands in the Command window to see how they affect the form. Note that the third command will terminate the form and you'll have to run the form again! You can waste the better part of an afternoon playing with the various properties, setting them to different values, turning buttons on and off, and so on. I know I have.

```
_screen.activeform.command1.visible = .f.
_screen.activeform.command1.visible = .t.
_screen.activeform.release()
_screen.activeform.text1.height = 80
```

7. Close the form by clicking the Close box in the upper right side of the title bar or select the Close menu command from the Control menu (click on the icon in the upper left side of the title bar).

## Calling methods from objects on the form

You can now make changes to the form's controls from within the form. You've already done this when you've changed the caption of a command button or calculated a new value to be displayed in a text box. But you also have access to the events and methods of a form, and its controls as well. It's probably a good idea to make clear, again, the distinction between a method and an event, and the example I'll present here will emphasize the difference.

An event is an action that automatically happens. For example, when you click a command button, the Click event is fired automatically. Or, when you tab into a text box, the GotFocus event fires. You don't have to do anything for either of these events to occur. A method, on the other hand, is a procedure—a subroutine of code that you write. You create a procedure of your own, place the method code into that procedure, and then explicitly call that method, or you can attach your own method code to an existing event. I'll show you how to do both.

You'll put several simple controls on a form, then add code to several of the existing events—both of the form and of the controls you've added—and then add your own procedures and call them.

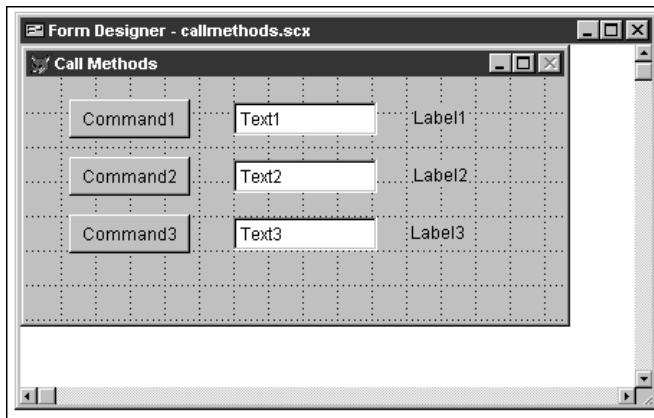
1. Create a blank form.

- Change the following properties:

**Caption: Call Methods**  
**Name: CallMethods**

- Save the form with the file name "CallMethods".

2. Place three command buttons, three text boxes, and three labels on the form as shown in **Figure 8.4**.



**Figure 8.4.** The Call Methods sample form.

3. Enter the following code into following form events:

```
Load():
messagebox("Form Load")
```

```
Init():
messagebox("Form Init")
```

```
Show():
messagebox("Form Show")
```

```
Activate():
messagebox("Form Activate")
```

4. Enter the following code into the following events of each control:

```
command1.click():
messagebox("Command1 click")
thisform.YourCustomMethod()
```

```
command2.click():
messagebox("Command2 click")
thisform.command1.click()
```

```
command3.click():
messagebox("Command3 click")
```

```
text1.GotFocus():
thisform.label1.caption = "Text1 got focus"
```

```
text1.LostFocus():
thisform.label1.caption = "Text1 has now lost focus"
```

```
text2.GotFocus():
thisform.label2.caption = "Text2 got focus"
```

```
text2.LostFocus():
thisform.label2.caption = "Text2 has now lost focus"
```

```
text3.GotFocus():
thisform.label3.caption = "Text3 got focus"
```

```
text3.LostFocus():
thisform.label3.caption = "Text3 has now lost focus"
```

5. Create a brand new method of your own, called YourCustomMethod.

- Open the Form menu and select NewMethod.
- Enter “YourCustomMethod” in the Name text box. (Note that whether you use uppercase, lowercase, or “camel” case is irrelevant because VFP will convert the name to lowercase automatically. I just used camel case because it’s easier to read in the book.)
- Click the Add button.
- Click the Close button (because the New Method dialog stays open after you add a method).

- Find the YourCustomMethod method in the Methods tab of the Properties window. It's at the very bottom of the list. Note that all custom properties and methods are located after all of the native Visual FoxPro properties and methods.
- Open the method, and add the following code:

```
messagebox("Called from YourCustomMethod")
```

6. Run the form.

- You'll notice that you'll get four message boxes, in the order of Load, Init, Show, and Activate, before the form actually displays. Once the form displays, you can select another window, such as the Command window. When you select the form again, the Activate message box will display again, because the Activate event is firing when you make the form active.
- Tab through the form, and watch each of the labels change as you move into and out of each text box.
- Finally, click each command button. You'll get two message boxes when you click Command1: one when the Click event fires (and thus displays the message box in response to the command), and one that indicates YourCustomMethod has been called from the Click event.

Clicking the second command button will display three message boxes: one for the Command2 Click event, and then the two that were also fired when the Command1 Click method was called. It's important to note that you're simply executing the code in the Command1 Click method—but you are not actually firing the Click event itself.

If you're tempted to brush over this concept, don't! A number of things happen when you click a command button, and one of them is the Click event firing. Another is that if the command button didn't have focus before you clicked it, it gets focus when you click it. You can prove this by putting the following code in Command1's GotFocus event:

```
messagebox("Command1 GOT focus")
```

When you run the form, you'll get the four message boxes for Load, Init, Show, and Activate. Then you'll get a fifth message box because the first control on the form—Command1—then gets focus, and the message box is displayed.

If you click on, say, Command3, just to move focus away from the action, and then you click Command2, you'll get three message boxes: one for Command2's Click event, and then the two that are called from Command1's Click event. Note that even though Command1's Click event was fired, the GotFocus event for Command1 was not—because calling Command1's Click event is not the same as clicking the button itself.

## Integrating forms and data

Now that you're comfortable building a form (you *have* spent a few afternoons goofing around with this stuff, haven't you?), manipulating properties, and getting data from the user, it's time to look at getting a form to talk with data. In this example, I'm going to build a simple form that will navigate through a table. If you've used previous versions of FoxPro, you'll notice that SCATTER and GATHER are no longer part of the technique.

1. Create a blank form.

- Change the following properties:

```
Caption: People  
Name: People
```

- Save the form with the file name "People".

2. Do NOT put labels or text boxes on the form yet!

3. Put two command buttons on the form.

- Change the following properties:

```
Command1:  
Caption: Next  
Name: cmdNext  
Command2:  
Caption: Previous  
Name: cmd Previous
```

- Add the following code to the methods:

```
cmdNext.click():  
skip  
thisform.refresh()  
  
cmdPrevious.click():  
skip -1  
thisform.refresh()
```

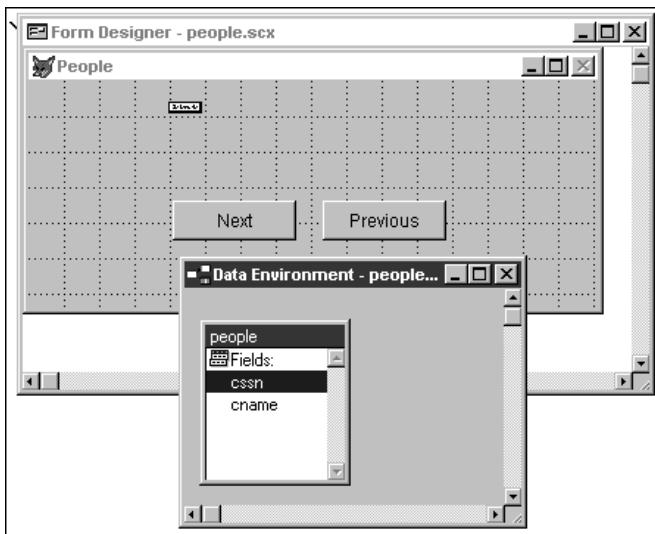
4. Attach data to this form by adding a data environment.

- Right-click on the form and select the DataEnvironment menu command.
- The Open dialog will appear, asking you to select a table or database. Select the PEOPLE table. A People window will appear in the Data Environment window. The Add Table or View dialog will still be displayed, so click the Close button.

5. Add data-bound controls to the form from the data environment.

- Click one of the fields in the People window in the Data Environment window, and drag it to the form, as shown in **Figure 8.5**. When you drop the field on the form, a text box and a related label will be added to the form. The label's caption

will be the name of the field in the table, and the text box will be sized approximately to the size of the field.



**Figure 8.5.** Dragging the CSSN field from the People table in the Data Environment window to the People form.

- If you then examine the text box, you'll see that several properties have been automatically set as a result of dragging the field from the data environment to the form. For example, the ControlSource has been set to the table.fieldname value, the MaxLength property has been set to conform with the length of the field in the table, and the text box has been named in accordance with the standard naming conventions described in online help, such as txtcSSN if you dragged the cSSN field.
6. Run the form.
- Click the Run button.
  - Clicking the Next and Previous buttons will move the record pointer through the table one record at a time. If you go too far in either direction, you'll get an appropriate error, because there isn't any error trapping in the simple code in the two command buttons' Click events.
  - Now try changing the contents of some of the fields. Type in a new value and then move on to the next record. Moving back to the original record shows that the change was indeed written to the table. A number of questions immediately crop up. The first one concerns the fact that we never wrote a single line of code to handle opening the database or PEOPLE table. This is all handled behind the scenes via the Data Environment. The second question that many of you Xbasers

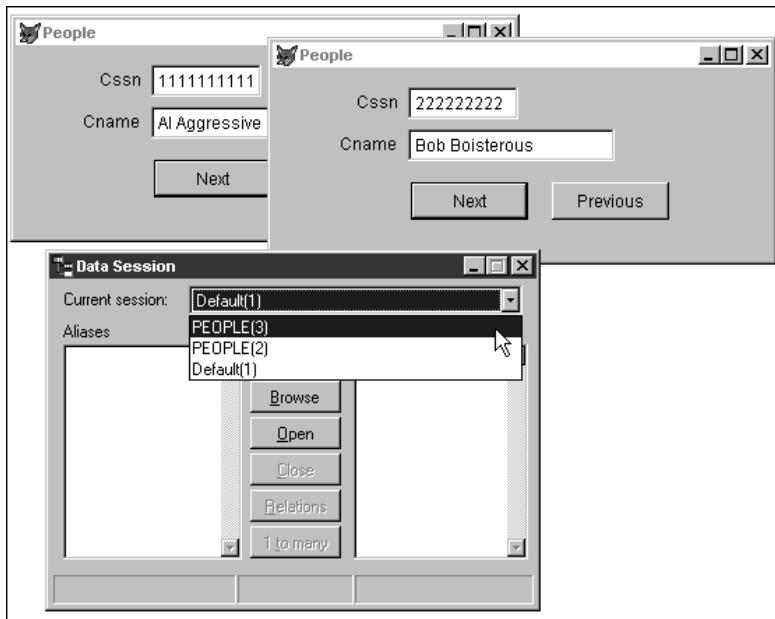
are asking is, “How are you getting data from the table to the form, and how are you getting the changes back to the table?” The answer is simply that each control is bound directly to the field in the table—much like you could do with direct editing in FoxPro 2.x.

- Visual FoxPro contains a new mechanism, buffering, for moving data between forms and tables if you don’t want to use direct editing. I’ll discuss the ramifications of data buffering in detail in the next section of the book, including multi-user concerns and handling contention.
7. Try running a second copy of the form.
- After running the form once, click in the Command window and issue the command DO FORM PEOPLE. If you’ve been playing around while I was looking somewhere else, you might have accidentally already run a second copy of the form. It might not have been obvious, because the second copy will reside on the same space as the first. If that’s the case, drag one copy of the form away so that you can see both.
  - As you navigate between records in one form, nothing will happen in the second form. However, it’s simply because the display has not changed. Try this:
    - Move to the first record of the first copy of the form, and then press Next, so that you’re on record number 2.
    - Click on the second form, note which record you’re on, and then press Next. You won’t go to the record following the one that you had been looking at—instead, you’ll move to record number 3. Both forms are simply different windows looking at the same record pointer in the same table.

Rather disappointing—almost thought we were going to be able to play some handy tricks, eh? Ha! All is not lost.

8. Create private data sessions.
- Close the forms, close the PEOPLE table, open the PEOPLE form in design mode, and change the form’s DataSession property to 2 – Private Data Session.
  - Run two or more copies of the form. To run more than one copy, either issue the command DO FORM PEOPLE or click the Run button to get one copy up and running. Then, while that form is active, notice that the Command window displays again. Click in the Command window and issue the command DO FORM PEOPLE a second time. A second instance of the form appears on top of the first one. Move it to another location on the screen. You can now move between forms just as you would with any other two windows on the screen. Navigate through both copies of the form. As you switch between the forms, notice that you truly have two unique views on the table, much as when you have two users on different computers using the same application—both can do their own work, even on the same table, without colliding with each other.

- In other words, you can think of this “private data session” property as having two copies of the same form—each running by itself. Each copy of the form has its own “copy” of the table that it is working with—much like several users on a network.
- If you open the Window menu, you’ll see two instances of the PEOPLE form, and if you open the Data Session window, you’ll see two instances of the PEOPLE form available for switching between, as shown in **Figure 8.6**. The second and third names are the names of each data session, just as you can USE a table AGAIN with a different alias.



**Figure 8.6.** Setting a form’s *DataSession* property to “2 – Private Data Session” enables you to run two or more independent copies of a form.

## Up close with the Form Designer

There are several components in VFP that you’ll use when designing forms. Here, I’ll walk you through each component individually.

### Form Designer

The Form Designer consists of two windows. The larger one—the Form Designer window—displays the entire available screen. In other words, if you’re working with a 640 x 480 screen, you can move around in the Form Designer window and see the entire surface. The available surface and the area beyond are distinguished by different colors. The smaller window shows

you the size of the form you are creating. You can't create a form larger than the screen size that is set in the Forms page of the Tools, Options dialog. When you create a formset (two or more forms that work in concert), all of the forms in the formset will appear in the Form Designer window. It is possible to have a formset where all of the windows will not fit nicely on the screen—one will have to overlap the other. However, even with a formset, no individual form can be larger than the screen.

Right-clicking in the Form Designer window will open a context menu with the following choices:

<b>Run Form</b>	Runs the form. If you have checked the "Prompt to save changes before running form" check box in the Forms tab of the Tools, Options dialog, you will be prompted to save any changes. If the check box was not checked, the form will automatically be saved before it is run.
<b>Paste</b>	Pastes to the form a copy of whatever is on the clipboard as appropriate.
<b>Data Environment</b>	Opens the Data Environment window.
<b>Properties</b>	Opens the Properties window.
<b>Builder</b>	Runs the currently registered Form Builder. See Chapter 20 for more about Builders.
<b>Code</b>	Opens the Code window for the form.

Right-clicking an object on a form will bring up a shortcut menu with the following menu options:

<b>Undo</b>	Cancels the most recent action, such as resizing or moving the object, or changing a property.
<b>Cut</b>	Deletes the selected object or objects and places them on the clipboard.
<b>Copy</b>	Copies the selected object or objects to the clipboard.
<b>Edit</b>	If the selected object contains other objects (like a PageFrame containing multiple pages), enables the objects inside the selected object for selection themselves.
<b>Properties</b>	Opens the Properties window.
<b>Builder</b>	Opens the builder for the selected object. Note that if more than one object is selected, the builder will apply to all of the objects if possible.
<b>Code</b>	Brings forward the Code window for the current object.

## Properties window

The Properties window allows you to view and edit the properties and methods attached to any object in the Form Designer, including the formset, each form, containers, and controls.

The Object combo box at the very top of the window allows you to select an object in the Form Designer, and the Page Frame below it will then display the properties and methods for that object. The objects in the combo box are listed in hierarchical order: The formset, if any, will be listed first, then the first form, any container objects in that form, then control objects, and so on for each additional form. Within a hierarchy, the objects are listed in the order that you created them.

Right-clicking on the title bar of the Properties window displays a context menu that allows you to set a variety of properties for the Properties window. See **Figure 8.7**. Each of the first

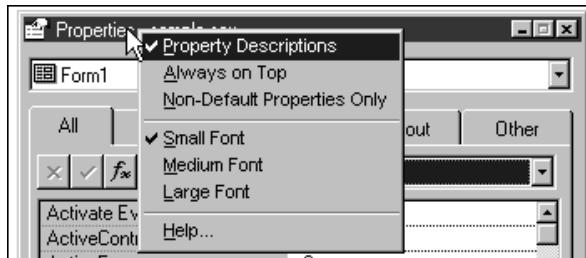
three menu commands can be selected (or deselected) simply by clicking them. Checking Property Descriptions will cause a read-only edit box to display at the bottom of the Properties window—I always leave this on unless I'm really cramped for room on the screen.

Checking Always on Top will cause the Properties window to float on top of other windows, even when it doesn't have focus. This can be handy to avoid losing the Properties window underneath a dozen other windows, but if you're short on screen space, it can also be annoying because you'll be constantly moving it around as you're trying to work with whatever's underneath. Remember that you can always tell which window is active because its title bar will be a different color than the rest of the windows on the desktop.

Checking Non-Default Properties Only will display only those properties that have been changed from their default values, and methods that have code added to them. I typically keep the Properties window open and sized to fit the entire height of the screen, so I can see as many properties as possible at a glance. This option, then, is terrifically handy when you are working with a series of objects that each have hundreds of PEMs (Properties, Events, and Methods) and you only want to change a couple of properties of each, and you don't have the screen space to keep the Properties window open all the way.

The second group of menu commands works in concert—much like an option button group. The selection you make determines the size of the font used in the list box that displays all of the properties and methods, but it doesn't change the font of the Object combo box, the tabs of the page frame below the Object combo box, or the text box or combo box that displays the value of the currently selected property.

In addition to seeing a PEM's Property Description displayed for the highlighted PEM, you can select the Help menu command or press F1 to display the help topic for that PEM.



**Figure 8.7.** Right-clicking on the Properties window title bar will display a useful context menu.

Underneath the Object combo box is a five-tab page frame, as shown in **Figure 8.8**. The first tab, All, displays all PEMs for the currently selected control. The next four display just a subset of PEMs—the Data tab displays those properties that have to do with data binding of an object to its data source; Methods displays a list of all events and methods for the object; Layout displays those properties affecting the physical appearance of the object; and Other displays the remaining properties. Naturally, different objects have different properties, events, and methods, and so some of these tabs might be empty or sparsely populated.

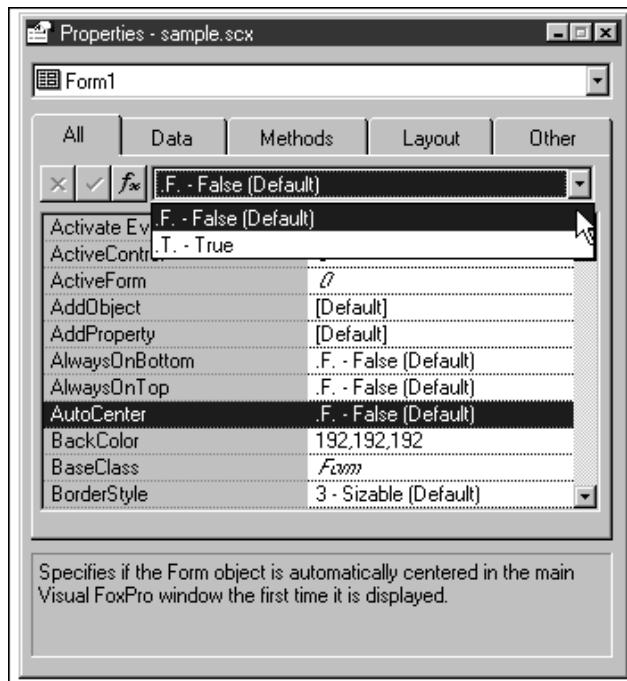
The set of controls immediately underneath the tabs is called the Property Settings box; it contains four controls that allow you to change the value of the currently selected property.

The list box below the Property Settings box displays a list of the properties or methods according to which tab is selected. You can navigate through this list box to select a specific property with which to work.

The type of control on the far right of the Property Settings box varies according to the type of property that is highlighted in the list box. Some properties can have free-form text values, such as the caption of a form, and so the control is simply a text box. Other properties have predetermined values, so the control is a combo box with the allowable choices already provided. For example, the Enabled property can be set to True or False. Still other properties can have a large number of choices, such as the foreground or background color. The control for these is a combination of a text box and a command button, called a dialog button, which allows you to open a dialog to select the property's value from a dialog box.

When a method is highlighted, the controls in the Property Settings box are disabled.

The command buttons on the left side of the Property Settings box are, from left to right, Cancel, Accept, and Expression Builder. Once you have entered a value for a property, click Accept to confirm the value, or click Cancel (or Escape) to abandon your change and revert to the previous value. The Expression Builder button opens the Expression Builder so you can create a complex expression.



**Figure 8.8.** The five tabs in the Properties window allow you to view some or all of the PEMs for an object.

The list box of properties is extremely robust and contains a lot of hidden functionality.

Properties that are read-only at design time are displayed in italics. Properties that you have changed are displayed in bold.

You can place your cursor between the two columns and resize the columns to see either a long property name, or the value of a property that extends past the edge of the second column. If that's not enough, of course, you can widen the Properties window itself.

Pressing Ctrl+Alt plus the first letter of a property name will move the highlight down to the first property that begins with that letter. Subsequently pressing that letter (while holding down the Ctrl+Alt key combination) will move down one property at a time.

You can double-click a logical property to change the value from True to False or vice-versa, and on a multi-valued property to cycle from one value to the second to the third and so on. You can also double-click on a color-related property to bring up the Color dialog.

Right-clicking a property in the list box will bring up a context menu whose options vary according to the type of property. Clicking Reset to Default will change the property back to its original value. This is different than simply manually changing the property back to its default value. Here's why. Like earlier versions of Fox, Visual FoxPro forms are stored on disk as tables—if you created a form named SAMPLE, you could look at the table by using the commands:

```
use SAMPLE.SCX
browse
```

One of the fields in this table is a memo field that contains the values for each property that was manually changed. Thus, if you changed the Caption and Enabled properties, the Properties memo field of SAMPLE.SCX would contain the following information:

```
Caption = "My Test Form"
Enabled = .F.
```

If you then changed the Enabled property back to True by double-clicking it, the property would be bold, indicating a manual change, and the contents of the Properties memo field of SAMPLE.SCX would look like this:

```
Caption = "My Test Form"
Enabled = .T.
```

If, on the other hand, you just Reset to Default the Enabled property, the contents of the Properties memo field would look like this:

```
Caption = "My Test Form"
```

The Zoom menu command on the context menu will display a small editing window that's easier to use for editing long expressions than the limited area in the text box. The Expression Builder menu command will open the Expression Builder, of course.

## Code window

The Code window is used to create and edit methods for an object's events. It consists of two combo boxes and a text-editing window. The Object combo box allows you to move between the various objects in the form (or formset), and the Procedure combo box allows you to move between the various methods of a specific object.

Given the number of objects that a form could contain, as well as the number of events and methods that each object has, you might be wondering how you're going to remember which objects and methods contain code.

First of all, every method that contains code is displayed in bold, with a value of User Procedure in the Properties window. Second, when you open the Procedure combo box in the Code window, those methods that have code will be displayed in bold—and they will be moved to the top of the list. Even better, however, is the functionality of the PageUp and PageDown keys. With both combo boxes closed, press either key. The code window will automatically jump to the next combination of object and procedure that contains custom code. As a result, you can easily navigate from one procedure to the next that contains code.

Right-clicking on the Code window will not bring up a context menu.

## Form Designer toolbar

The Form Designer toolbar acts as a launching pad for quickly getting at various Form Designer tools. It contains the following buttons, from left to right (see **Figure 8.9**):

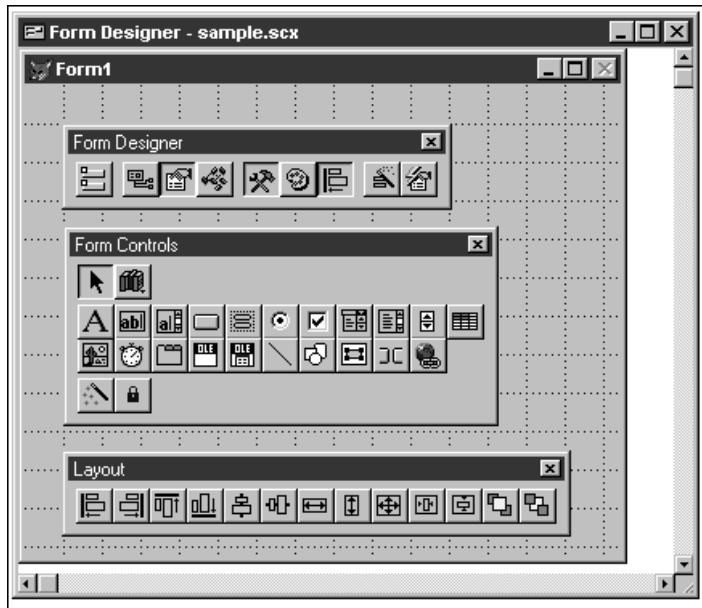
<b>Set Tab Order</b>	Changes the form to Set Tab Order mode.
<b>Data Environment</b>	Opens the Data Environment window.
<b>Properties window</b>	Opens the Properties window.
<b>Code window</b>	Opens the Code window.
<b>Form Controls toolbar</b>	Opens the Form Controls toolbar.
<b>Color Palette toolbar</b>	Opens the Color Palette toolbar.
<b>Layout toolbar</b>	Opens the Layout toolbar.
<b>Form Builder</b>	Allows you to select tables, create forms, and apply styles of fields, including elegant, contemporary, or professional.
<b>Auto Format</b>	A builder that allows you to select from one of three default formats for specific properties of all of the objects on the selected form, including borders, colors, fonts, layout, and 3-D effect.

I personally don't ever have the Form Designer toolbar up, because I usually keep open the tools that it launches instead.

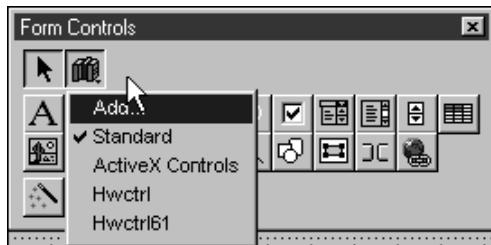
## Form Controls toolbar

The Form Controls toolbar, as shown in **Figure 8.10**, is used to place controls onto a form.

The first button on the top row, Select Objects, is used simply to deselect another button that has been depressed. The second button, View Classes, has a small arrow in the lower right corner. This signifies that clicking the button will open a menu. This menu allows you to choose which set of controls will be shown on the Form Controls toolbar. By default, the standard Visual FoxPro classes are displayed. You can change this to display selected ActiveX controls, or one of your own set of control classes as desired.



**Figure 8.9.** The Form Designer, Form Controls, and Layout toolbars.



**Figure 8.10.** Clicking the View Classes button displays a context menu of available sets of controls.

The middle two rows of buttons are for each native Visual FoxPro control. I'll discuss each of these at length in Chapter 9, so I'll skip over them now. The last row's buttons are the Builder Lock and the Button Lock. Clicking the Builder Lock will cause the builder for a specific control to be invoked once you place the control on a form, instead of placing the control on the form and then calling the builder from the control's context menu. You'll need to click the Builder Lock again to turn this functionality off.

When you click a control's toolbar button, it stays depressed until you place the control on a form, and then it automatically pops out again. This can be annoying if you want to drop several copies of the same control on a form. The Button Lock will keep the button for a control pressed in after the control has been dropped on a form. This is really handy—it allows

you to place several of the same type of control on a form simply by dropping them one after another on the form, instead of having to click the control's toolbar button, drop it on the form, click the toolbar button again, and so on. When you are done dropping controls on a form, click the Button Lock again.

## Layout toolbar

The Layout toolbar is used to manipulate controls on a form or a report, and contains the following buttons, from left to right (see Figure 8.9):

<b>Align Left/Right/Top/Bottom Sides</b>	The first four buttons will align all selected controls to the named position. In other words, if you put a bunch of text boxes on your form, select all of them and then press Align Left Sides, they'll all be left-justified to the left edge of the left-most control. If you hold down the Ctrl key while clicking on the button, they'll all be left-justified to the left edge of the right-most control.
<b>Align Vertical/Horizontal Centers</b>	Similar to the first four, these two buttons will align all selected controls along an imaginary vertical horizontal axis. I can never keep vertical or horizontal straight, so it's handy to have the picture on these buttons.
<b>Make Same Width/Height</b>	These buttons will resize all selected controls to the same width or height of the control with the largest dimension. Similar to the Align Left/Right/Top/Bottom buttons, holding down the Ctrl key while pressing this button will resize all selected controls to the smallest dimension.
<b>Make Same Size</b>	This button will resize all selected controls to the same size as the largest control in the selection. Holding down the Ctrl key at the same time will resize all selected controls to the same size as the smallest control in the bunch.
<b>Center Horizontally/Vertically</b>	These buttons will center a control or group of controls within the confines of the form they're contained in. Note that if a group of controls is selected, the entire group will be treated as one object.
<b>Bring to Front/Send to Back</b>	When you place a control on a form, it's as if you're placing a transparent overlay on the form. When you place a second control on the form, think of it as being placed on a second transparent overlay. If those two controls are on top of each other, it's possible for the first control to be obscured by the second one that's "on top" of it. You might want the first one on top of the second one. These two buttons allow you to rearrange the "order" of the "overlays" by pulling the selected control up to the top, or pushing it down to the bottom.

I keep the Layout toolbar open all the time, because I'm forever changing the visual appearance of the form, and thus want to change the way the controls are aligned.

## Menus

When the Form Designer window or Properties window is on top, the following menus change to include the following menu options. Not all menu options are available at all times. For example, when the Code window is on top, the View and Format menus change back to display normal text-editing menu options because you'll be doing text editing.

### View menu

The View menu contains the following additional menu options when the Form Designer window is active:

<b>Tab Order</b>	Allows you to change the order in which focus moves from control to control. Tab Order is covered in more detail later in this chapter.
<b>Data Environment</b>	Opens the Data Environment window.
<b>Properties</b>	Opens the Properties window.
<b>Code</b>	Opens the Code window.
<b>Form Controls Toolbar</b>	Displays the Form Controls toolbar.
<b>Layout Toolbar</b>	Displays the Layout toolbar.
<b>Color Palette Toolbar</b>	Displays the Color Palette toolbar.
<b>Grid Lines</b>	Makes grid lines on the form either visible or invisible. This overrides the setting of the Grid Lines check box in the Forms tab of the Tools, Options dialog. The distance between grid lines on a form is controlled by the Horizontal spacing (pixels) and Vertical spacing (pixels) settings in the Forms tab of the Tools, Options dialog. You should leave this on unless you've got perfect vision out of all seven eyes and thus can line up controls without any aid whatsoever.
<b>Show Position</b>	Displays or hides the location and size of the current control in the right side of the Status Bar. This overrides the setting of the Show Position check box in the Forms tab of the Tools, Options dialog. You should also leave this on, again, unless you have the perfect vision mentioned above. You think I'm joking? You might not be able to tell, or care, if a couple of your controls are out of sync with each other. But there are humans—lots of them, actually—who can discern this type of aberration, and it will ruin their day, much like the princess and the pea.

### Format menu

The Format menu has a number of choices you can use to simultaneously manipulate multiple objects on a form; it also has many additional functions.

<b>Align</b>	Includes the same choices as the Layout toolbar: Align to Left/Right/Top/Bottom, Vertical/Horizontal Centers, and Center Vertically/Horizontally.
<b>Size</b>	Includes a couple of new choices as well as some from the Layout toolbar. Fit will resize a control to the length of its caption—useful particularly for option buttons and check boxes. Grid will snap a control to the nearest grid lines. Tallest/Shortest/Widest/Narrowest work as they do on the Layout toolbar.
<b>Horizontal/Vertical Spacing</b>	Allows you to make the space between selected controls the same as the first two controls on the top or to the left, or to increase or decrease the space a pixel at a time.
<b>Bring to Front/Send to Back</b>	Performs the same functions as their counterparts on the Layout toolbar.

<b>Snap to Grid</b>	Allows you to determine whether or not controls that are placed on a form will be snapped to the grid on the form. The setting of the Snap to Grid check box in the Forms tab of the Tools, Options dialog determines the way this menu option is initially set. Whether or not you should leave this on is a topic hotly debated late at night at those programmer conferences—some people like it on, because you can nudge objects one pixel at a time using the keyboard arrows, while others are just control freaks (is that a great pun, or what?), and want to do it all themselves.
<b>Set Grid Scale</b>	Allows you to set the distance, in pixels, between grid lines on a form. The settings of the Horizontal/Vertical spacing (pixels) spinners in the Forms tab of the Tools, Options dialog determines the way this menu option is initially set. A lot of developers set this to a 5x5 or 6x6 grid for more granular divisions on the form.

## Form Menu

The Form menu allows you to manipulate the form and its objects. It contains the following options:

<b>New Property</b>	Allows you to create a new property for the form. (Before you go looking for this same function—and New Method, to boot—for a control, I'll stop you here. It's only available for forms and classes.)
<b>New Method</b>	Allows you to create a new method for the form.
<b>Edit Property/Method</b>	Allows you to modify and delete (using the Edit Property/Method dialog) a property or method you created using one of the first two menu options. See Chapter 10 for more information about user-created properties and methods.
<b>Include File</b>	Specifies an include file for the form. Include files are covered later in this chapter.
<b>Create Formset</b>	Creates a formset object. This is required before adding additional forms.
<b>Remove Formset</b>	Deletes the formset object (available only when there is a single form in the formset).
<b>Add New Form</b>	Creates an additional form within a formset.
<b>Remove Form</b>	Deletes a form from a formset.
<b>Quick Form</b>	Opens the Form Wizard.
<b>Run Form</b>	Runs the form after prompting you to save any changes.

## Tab order

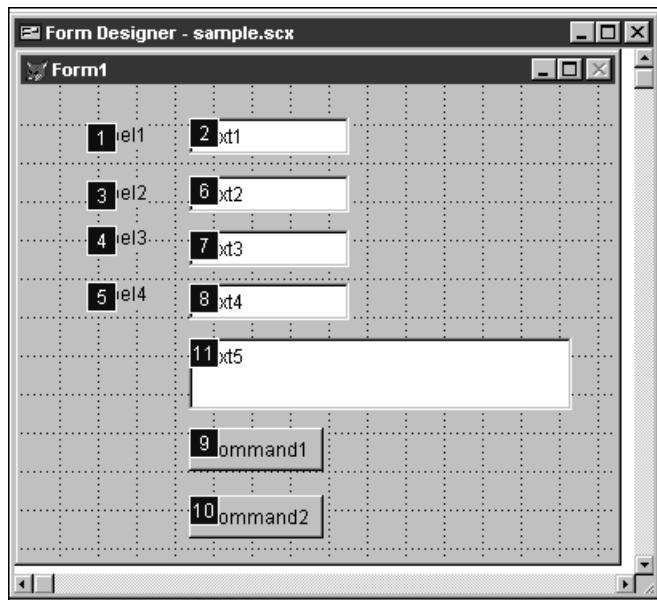
Tab order is the order in which the controls receive focus as the user is tabbing through the form. This is one of those features that's nearly an afterthought to a developer, but of critical importance to a user, because it can significantly help or hamper productivity. The native tab order of a form is the order in which the controls are placed on the form; it has nothing to do with their physical position on the form.

Using the Tab Ordering combo box in the Forms tab of the Tools, Options dialog, you can choose between two methods for rearranging tab order after placing controls on a form.

If you select Interactive from the Tab Ordering combo, then when the Form Designer window is active and you select the View, Tab Order menu option, you'll be able to click each control in the order you want them to receive focus. See **Figure 8.11**. Note that every control,

whether or not you might actually enter data in it, will participate in the Interactive Tab Ordering process.

In Figure 8.11, a numbered label is attached to every control on the form, reflecting the current tab order. To change it, start clicking on the numbered labels in the desired tab order. You can ignore all of the labels and any other controls that won't participate, but be careful—if you make a mistake, it's difficult to recover and you'll probably find yourself starting all over again. This can be a daunting proposition if you've got 50 or 60 controls on a form, so it's a good thing there's a second way: By List.

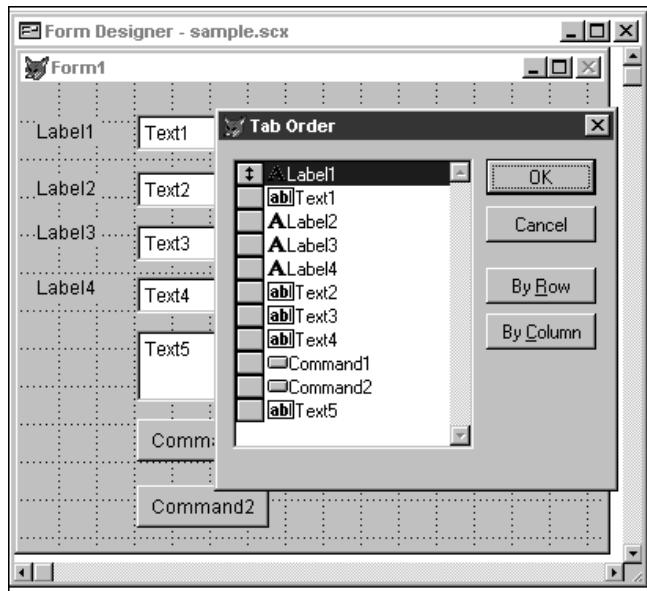


**Figure 8.11.** “Interactive” tab ordering allows you to visually select the order in which focus moves from control to control.

If you select By List from the Tab Ordering combo, you'll be able to use a different mechanism when selecting the View, Tab Order menu option. The dialog that appears, as shown in **Figure 8.12**, presents a couple of ways to redefine the tab order. If you click the By Row or By Column buttons, all of the controls on the form will be ordered according to their physical position on the form.

The controls are all listed in the list box in the Tab Order dialog, and you can use the mover bars to the left of each control to position each control precisely in the tab-ordering scheme.

You might find it handy to click the By Row button first, to get a rough pass at the order, and then use the mover bars to fine-tune any adjustments as needed.



**Figure 8.12.** By List tab ordering allows you to manipulate objects through mover bars in a list box.

## Include files

In Chapter 2, I showed you how to define a constant using the #DEFINE and #UNDEF keywords; these are handy if you use them in a program file. If, however, you want to use constants in a form, you'll run into trouble, because a constant is available only within the event or method in which it's used, and that's not going to be very helpful because you'll probably want to have the constant available throughout the form.

You can create a separate text file, called a header file, that contains all of your #DEFINE statements, as shown in the following code. Then you specify this header file as a form's include file, and the constants will be available throughout the form. To do so, open the Form menu and select Include File, and use the Include File dialog to select the header file you want to add to your form. Note that include files traditionally have an extension of .H (for "header").

It might be inconvenient to specify the same include file for form after form after form. You can define a default header file in the Forms tab of the Tools, Options dialog. This default will be overridden by any choice you make in the Include File dialog. You can also use the \_INCLUDE system variable in a config file—this is covered in Chapter 16.

```
--- General
---ZOrder Method
#DEFINE ZORDER_BRINGTOFRONT 0
#DEFINE ZORDER_SENDOBACK 1

--- TYPE() tags
#DEFINE T_CHARACTER "C"
#DEFINE T_NUMERIC "N"
#DEFINE T_DOUBLE "B"
#DEFINE T_DATE "D"
#DEFINE T_DATETIME "T"
```

## Referencing objects

A key skill in using the Form Designer is being able to manipulate the various objects associated with a form. The hierarchy is rather straightforward, but the syntax takes a little getting used to.

### Referencing objects by their full names

Just like referencing a specific file on a hard disk, the complete name for an object consists of a fully qualified string of the names of its container hierarchy. For example, imagine a control in a specific page of a page frame in a form within a formset. The fully qualified name for the control must include the formset, form, page frame, page, and, finally, control name. To reference a property of a control, separate the names of the objects in the hierarchy by periods, starting with the largest object and continuing until you reach the control. For example, the value of the Caption property of the Done command button in the Enter Name form that is part of the Search formset would be determined by the expression:

```
frsEnterName.frmSearch.cmdDone.Caption
```

You can use this expression in several ways. You can use it to determine the value of the property, or you can assign a value to the property. This bit of code does both:

```
if frsEnterName.frmSearch.cmdDone.Caption = "Label"
  frsEnterName.frmSearch.cmdDone.Caption = "Done"
endif
```

A form is called a “container” because it can “contain” other controls. Thus, your naming scheme goes from the outermost container to the innermost. A form isn’t the only type of container, however. Formsets are containers, of course, because they contain one or more forms. Other examples are page frames, option button groups, grids, and command button groups. Suppose you had a button on page 2 of a page frame. You would reference the caption like so:

```
frmMyForm.pgfMyPageFrame.pagTwo.cmdMyButton.Caption
```

If you had an option button group on page 3 of that page frame, you'd reference the caption of the fourth option button in the group like so:

```
frmMyForm.pgfMyPageFrame.pagThree.opgMyGroup.opbButtonFour.Caption
```

There are three variations of this syntax. The first allows you to reference an object when you know its full name, as I've just covered. The second allows you do so when you know the hierarchy, but not specifically which form the object is in. And the third allows you to reference an object when you don't even know the hierarchy, but you do know where the object is in relation to another object.

### **Referencing objects by their container**

I just mentioned that you might need to reference an object but might not know which form it's contained in. How in the world could that happen?

Sometimes you might need to reference a control in the form, but might not know which form you're in, just as you might need to reference a file in the WINNT directory, but not know whether WINNT is on drive C, D, or R. You can start from the "root" of the current drive, such as \WINNT\SYSTEM32\MyFile.txt, and you can start from the "root" of the current form, like so:

```
thisform.cmdPost.caption
```

The term "thisform" allows you to refer to a form when you are in the form, but don't know its name. The term "this" allows you to refer to any object without knowing its name. Now, why would you be working with something when you didn't know its name?

Here's an example. First, it seems obvious that you're going to have to know something's name in order to change something about it. Suppose you've got a form with a Save command button. You're going to want to disable the Save command button until the user has made a change—providing a visual clue to the user as to the state of the system. In this case, the Enabled property will start out as .F., and when the user makes the change, you're going to want to change the Enabled property of the button from .F. to .T. Once the user has made a change, you'll change the property back to .F.

Thus, you'll need to know the name of the command button. Depending on where this change is being made—on the form level, the formset level, or another level in the hierarchy—you might need to reference the command button's parent—a page frame, a form, or a formset—and other objects in the hierarchy as well.

Now go a step further. Suppose (since I haven't gotten into the object-oriented capabilities of Visual FoxPro yet) you use a generic routine throughout your applications that checks for whether or not the user made changes. The routine then changes the Enabled property of the command button as needed. However, because you're using this same routine on multiple forms, you don't necessarily know the name of the form. Thus you can't correctly address the command button with a generic routine, because the full name of the object will change. However, you do know the name of the object—the command button—relative to the form, regardless of which form you're on. How? By using the "thisform." construct!

### Indirect object references

But it gets better, folks! You can use relative addressing to refer to objects, just as you can reference a file in another directory on the same level by using relative pathing (`..\otherdir\samefile.xxx`) instead of spelling out the whole file name. The “parent” keyword references the object that an object is sitting on. Thus, the syntax:

```
this.parent.cmdPost.Caption
```

refers to the caption on the command button “cmdPost” that’s on the same parent object as whatever this command is in—without knowing what the parent object is! This means that you could create a reusable group of objects and they could reference each other without knowing what form they were in—or even if they were in a form, a formset, or perhaps on a page in a page frame.

Another tricky point of syntax is accessing the value of a control. If you’ve used earlier versions of FoxPro, you’re used to using the name of an object as its value, so the following syntax would seem to make sense:

```
_screen.activeform.txtColor1 = 128  
_screen.activeform.txtColor2 = 0  
_screen.activeform.txtColor3 = 255
```

However, the value of a control—in this case, a text box on the form—is just another property of the control, as is the color or the caption. The property is named Value and you reference it like so:

```
_screen.activeform.txtColor1.Value = 128  
_screen.activeform.txtColor2.Value = 0  
_screen.activeform.txtColor3.Value = 255
```

## Tips and tricks

Oftentimes you will want to reference an object without knowing its explicit name. For example, during some event, you will want to address the active object on the active form, but you won’t know which object—indeed, even which form—is active. Or you might want to address a specific property of the current object, but you’re using a common method that will be applied to multiple instances of the object, so you won’t know the name for this specific object. How do you handle these situations?

## Referencing generic objects

Visual FoxPro has a set of properties and keywords that make it easier to reference an object in the hierarchy without knowing its name. The fundamental object in an application is the application’s screen, which has the name `_screen`. One or more formsets and forms might be within the screen at any one time. A formset can be referenced by the `ThisFormSet` keyword anywhere within the formset. A formset can contain one or more forms, and a screen can contain a form as well. The current form is referenced by the `ThisForm` keyword from within

the form and by ActiveForm elsewhere. Controls within a form are referenced by the This keyword from within the control and by ActiveControl elsewhere.

To try this, run a formset such as TWO\_SRCH. Then, place the following expressions in the Debug window and manipulate the formset by clicking on each form and tabbing through the various controls.

```
_Screen.ActiveForm  && shows "(object)"  
_Screen.ActiveForm.Caption  && shows "Search for Someone"  
_Screen.ActiveForm.ActiveControl  && shows "(object)"  
_Screen.ActiveForm.ActiveControl.Caption  && shows "Done"
```

The first example might be a little confusing. When you run a form, you are creating an instance of the form object, and this expression shows that the form has indeed been instantiated. When you close the form, the expression's value goes away, indicating that the object is no longer instantiated. Think of this as a variable that goes out of scope when you return from a subroutine.

## **Names and captions**

Note particularly that an object's caption is not the same as the object's name! You must reference an object by name or by reference, not by caption.

Use the Name property for controls to define names that make sense to you. These names are displayed in the Object combo boxes in the Properties and Code windows. The naming convention used in this book follows the recommended industry-standard guidelines.

To use the value of a variable as a caption, define the variable outside the form and then:

```
* this is in a PRG file that calls the form  
m.cXX = "My New Caption"  
  
* this is the caption property of a control in the form  
=m.cXX in the caption property
```

## **Code windows**

You can open multiple Code windows instead of flipping through the various objects in one Code window via the combo boxes. However, once you do so, you won't be able to access the code of a method that's already opened in another Code window, either by selecting it in the Object or Procedure combo box, or through the Find dialog. That method's name will appear dimmed in all other Code windows as a visual clue that it's already open.

You can double-click the name of a method in the Properties window to automatically open the Code window. Note that doing so will open up a second instance of the Code window!

## Layout

Get used to keeping the Layout toolbar open when using the Form Designer. And don't forget that there are additional layout menu options under the Format menu option in the main menu. I find the Vertical Spacing, Make Equal menu option to be especially handy.

The number of controls selected in the Form Designer window determine which menu options and layout icons are enabled in the Layout toolbar.

The positions of the various windows are stored in the FOXUSER resource file. If you keep the resource file turned off, you'll have to constantly rearrange your windows.

You can cut and paste controls and sets of controls from one form to another. Remember to adjust the properties of the controls as necessary—for example, you'll likely want to change the ControlSource for a text box.



# Chapter 9

## Native Visual FoxPro Controls

A form by itself doesn't do much. You've got to put stuff on it, and most often, that stuff is one or more controls. Visual FoxPro comes with a set of controls that you may well find meets the majority, if not all, of your needs. In this chapter, I'll discuss how to use each of VFP's native controls.

There are 21 controls native to Visual FoxPro, and among them they have 839 properties, events, and methods. Well, maybe 841 by now. Trying to examine each one of these controls and their PEMs in this chapter would be as unreasonable an exercise as including a 50,000-word dictionary in the third lesson of an English grammar text. Instead, I'm going to cover the key properties, events, and methods for each control, and show you how to use them in a simple form.

I'll cover more practical, real-world use in Chapters 12 through 15, and you can explore each PEM in depth in the *Hackers' Guide to Visual FoxPro 6.0* by Tamar Granor and Ted Roche.

### The proper care and feeding of controls

Controls can be divided into three groups according to their function: those that perform visual delineations, those that initiate actions, and those that allow you to manipulate data.

The visual delineators include the Label control, the Line and Shape controls, the Image control, and the Page Frame. These don't actually do anything, but they provide a means to provide visual feedback to the user.

The action controls include the Command Button and Command Button Group, and the Timer. You'll often see Windows applications that use other controls, such as check boxes, to initiate actions. In the discussion following this section, I'll explain why I believe this is a bad choice.

The data-manipulation controls include the Text Box, the Edit Box, the Option Group, the Check Box, the Combo Box, the List Box, the Spinner, and the Grid. These are used to accept input from the user, either to control the application or as a means to enter data into the application's tables. Many of these controls can be used for the same purpose—for example, you can use a text box or an edit box to enter a string of text into a field, and you can use a check box or a two-button option group to select between two choices. However, there are advantages to using certain controls for specific data-entry mechanisms, and I'll cover these in the section for each individual control.

The ActiveX and ActiveX bound controls are simply placeholders for third-party controls, and those could belong to any of the three groups.

### Focus and containers

The concept of "focus" is core, both to a form and to the controls placed on it. Focus is the GUI equivalent of a cursor in a character-based system. Focus is where the action is occurring on the

form; it might be a command button, an option group, or a text box. In much the same way as you can manipulate the cursor in a character-based system with the arrow and tab keys to move the cursor from field to field, you can manipulate the focus on a form by using either the keyboard (the arrow and tab keys) or the mouse (clicking on a control). A control changes its appearance when it has focus. Typically, you'll see the cursor blinking in a text box or edit box and you'll see a dotted line around the caption in controls like option groups, check boxes, or command buttons.

It's important to remember that, unlike many character-based systems, the Enter key does not usually change focus. Instead, if there is a default control on the form, pressing Enter executes that default control. Many times this is not what the user expects, so you'll need to account for this tendency.

One key point to remember about working with controls is that you must remember which level of the hierarchy you are in when editing. Some controls are simply containers for additional controls, such as the page frame, grid, option group, and command button group. It can be frustrating at first to try to manipulate the control inside a container and have Visual FoxPro prevent you from doing so. It is doubly frustrating to create a series of controls on a page of a page frame, only to find that you've actually placed all of the controls on top of the page frame itself—not in one of the pages.

Here's the trick. When you want to work within a container control, such as a page frame, place it on a form and then select the container. You'll see eight drag boxes—in the four corners and in the middle of each side. Then, right-click the container control to bring up the shortcut menu and select the Edit menu option. The container control will be highlighted with a thick green border, and you can now select individual controls in the container, such as an individual page.

An alternative method that works for all controls is to use the Object drop-down list box in the Properties window. You'll notice that the controls in the form are arranged in a hierarchical order with an outline format. You can scroll through the list box and select the desired object.

Some container controls have a predefined number of child objects but others do not. For example, when you drop a page frame on a form, it comes with two pages by default. Similarly, an option group comes with two option buttons. However, a grid does not come with any columns defined by default. You will have to set a property to define how many child objects you want—in the case of a grid, this property would be ColumnCount. Many controls have a number of properties and events in common, including Name, Caption, and BackColor, and events such as Click, Init, and Error. Instead of repeating the same information in each section, it makes sense to discuss handling these items just once.

## Common properties

### Colors

Controls have two color-related properties: ForeColor and BackColor. The former is the foreground color used to display text and graphics in an object, while the latter specifies the background color. For example, in a label, the ForeColor of the Visual FoxPro base class is black (the color of the text in the label) and the BackColor is gray.

The default BackColor property defaults to white for some controls, such as text boxes and edit boxes, and to gray for other controls, such as labels and check boxes. If your form has a

different BackColor, such as tan, the control will look out of place because there will be an invisible boundary with a white color. You'll likely want to change the color to match the form's BackColor. Note that this process can be made automatic by subclassing the controls and setting the BackColor property in the parent class. This is discussed in Chapter 10.

When you disable a control, such as a command button, the object automatically changes color so the caption uses the DisabledForeColor. You might want to be able to change the color of the command button's caption at other times as well, and you can do so with the ForeColor property. Be careful when doing so, because not all users have sensitivity to certain colors or schemes, and because it's easy to mistakenly turn the color of the button background and caption to the same color, making the button unreadable.

You can use the RGB() function to explicitly specify a color. The RGB() function takes three parameters, each a numeric value between 0 and 255, specifying the intensity of the Red, Green, and Blue color components. Experiment with the Color Picker to see the parameters needed for various colors. The following schemes can help get you started:

0,0,0	black
64,64,64	mottled red/pink
128,128,128	dark grey
192,192,192	light grey
255,255,255	white
128,0,0	magenta
0,128,0	light green
0,0,128	light blue
128,128,0	yellow
128,0,128	purple
0,128,128	light blue
255,0,0	bright red
0,255,0	bright green
0,0,255	bright blue
255,255,0	bright yellow
255,0,255	bright magenta
0,255,255	aqua

## Names and captions

The Name and Caption properties typically cause some confusion, because they seem to refer to the same property. The caption of a control is the actual text that shows up with the control on the form. Thus, some controls don't have a Caption property. For example, a check box's caption is the text that appears next to the check box itself. A text box, however, doesn't have a caption. The Name of the control, on the other hand, is the physical name of the control used by the developer to refer to the object within the program in order to manipulate it. For example, in order to determine whether a check box is checked, you'd need to evaluate the value of the check box, and you'd use the name of the check box to do so:

```
chkIsAlive.value.  
  
chkIsAlive.name = "chkIsAlive"  
chkIsAlive.caption = "Is Alive"
```

All controls must have a name in order to reference them, but some controls do not have captions that are displayed. For example, a command button group would have a name, but wouldn't need a caption because there is no visible object on the form for the group. Each command button, however, would have a caption in addition to a name.

It is easy to develop a set of names for controls that are difficult or impossible to remember, which makes development time-consuming, frustrating, and error-prone. The following conventions are suggested to help avoid these problems. (They can also be found in Help under the topic Reference, Language Overview, Language Categories, Naming Conventions, Objects.) First, use the following list of prefixes as the first three characters of an object's name:

acd	ActiveDoc
chk	CheckBox
cbo	ComboBox
cmd	CommandButton
cmg	CommandGroup
cnt	Container
ctl	Control
edt	EditBox
frm	Form
frs	FormSet
grd	Grid
grc	Column (within a Grid)
grh	Header (within a Grid)
hpl	Hyperlink
img	Image
lbl	Label
lin	Line
lst	List
olb	OLE Bound Control
ole	OLE Control
opt	OptionButton
opg	OptionGroup
pag	Page (within a PageFrame)
pgf	PageFrame
prj	ProjectHook
sep	Separator (within a ToolBar)
shp	Shape
spn	Spinner
txt	TextBox
tmr	Timer
tbr	ToolBar

The rest of the name of the control follows the prefix. In the case of a single control, such as a command button, the name might consist of the "cmd" prefix and the caption or the first word of the caption on the command button. In the case of a container, such as an option group, the name might consist of the "opg" prefix and a word that describes all of the option buttons together, such as Color or Season. It is customary to capitalize the first letter after the prefix, like so: cmdDone, opgSeason, tmrBegin.

If a control is bound to a field in a table, I'll use the name of the field as the name, like this text box that is bound to the last name of a person:

```
txtNameLast
```

Because Visual FoxPro will handle variable names and user-defined PEM names that are longer than 10 characters, I usually use the name of an accompanying descriptor for a container such as an option group. For example, I usually place a label near the option group to describe what the control represents, just as you would do in front of a text box. I'll use the label as the suffix for the name of the option group, like so:

```
opgRentalSeason
```

### AutoSize

AutoSize is another property that I set for many controls. This property automatically causes the control to be resized to the width of its caption at design time. Labels, option buttons, and check boxes are all likely candidates to have AutoSize set to .T. Many developers prefer to keep AutoSize set to .F. for command buttons—a form with several vertically aligned buttons that are all different sizes would look bad.

### Common events

All Visual FoxPro controls have Init, Destroy, and Error events. Most also share a number of other events, including Click, GotFocus, and LostFocus.

The Click event is fired when the user clicks the mouse while the mouse pointer is positioned over the control. All controls have a Click event associated with them, including those you wouldn't normally click on, such as labels and shapes. Think of the Click event of these types of controls as allowing the control to act as an invisible button. You can temper the allure of creating fancy and complex forms containing dozens of hidden Click events with the knowledge of the drudgery of educating users and maintaining a complex and less-than-perfectly-documented application.

The Init event is fired when the control is created. Note that a control's Init event is fired before the Init event of the corresponding container object (including the form itself!). In other words, you can think of Inits firing from the inside of a container to the outside. This is so that you can reference the control while creating the container.

The Error event of an object is fired when a runtime error is encountered in a method of that object. Suppose you have a form with a command button that has code in the button's Click event, and that code contains an error, such as trying to open a table that does not exist. To handle the error, place code in the command button's Error event. If you place code to handle that error in the form's Error event (as opposed to the command button's Error event), it will be ignored.

The GotFocus and LostFocus events are fired when the object receives or loses focus. The user can initiate the events by performing an action such as tabbing into or clicking on the control. The developer can also control the events with the SetFocus method.

You might want to move the focus from one control to another after an action has been

performed. (Fox 2.x users have been using \_curobj to do this.) For example, after clicking the Add button you would move the focus to the first text box in the form, so the user can begin entering data immediately. This is done with the SetFocus method. This is tricky, in that the call to the SetFocus method (remember, a method is simply a subroutine) is made in the event of the control you are leaving—not the one you are setting focus to. So in the Add button's Click event, you'd add the following code to move the focus to the Customer Name text box:

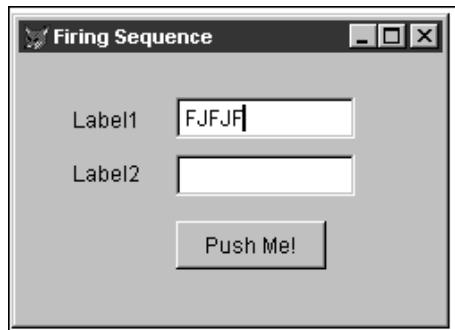
```
frmThisForm.txtCustName.SetFocus()
```

## Event firing

As you've seen, each control has a number of events associated with it. Much as it may seem to the contrary to a developer (particularly early in the morning with bleary eyes and tired fingers), these events do not fire at random occasions. Rather, they are executed in a specific order and in response to specific situations. To take advantage of an event's usefulness, it is best to be comfortable with the environment surrounding the firing of events.

This is one of those situations, much like riding a bike, where you can listen to someone else talk about it at length, but eventually you have to try it for yourself. Many programmers find it useful to create a sample form with code attached to each event of certain controls.

 The form FIRE, shown in **Figure 9.1** and included in this chapter's source code files at [www.hentzenwerke.com](http://www.hentzenwerke.com), contains several controls with commands that output a message. The message contains the name of the event to the Debug Output window in many of the methods. Run the form and see for yourself when each of the events is fired.



**Figure 9.1.** The sample form, FIRE, has messages in several of the events that are output when the event is fired.

Running this form, entering a five-character string in the first text box, and then tabbing through to the Push Me! button generates the output shown here:

```
Form.load()
label1.init()
text1.init()
label2.init()
PushMe.init()
text2.init()
```

```
Form.Init()
Form.show()
Form.activate()
text1.when()
Form.gotfocus()
text1.gotfocus()
text1.interactivechange()
text1.interactivechange()
text1.interactivechange()
text1.interactivechange()
text1.interactivechange()
text1.interactivechange()
text1.valid()
text1.lostfocus()
text2.when()
text2.gotfocus()
text2.valid()
text2.lostfocus()
PushMe.when()
PushMe.gotfocus()
Form.release()
PushMe.valid()
PushMe.when()
Form.destroy()
text2.destroy()
PushMe.destroy()
label12.destroy()
text1.destroy()
label11.destroy()
Form.unload()
```

You'll see that the form loads, but before anything else happens to the form itself, the controls are all instantiated (created). Then the form's Init, Show, and Activate events are fired. Third, the first control, text1, has its When event fired, and then the form and the text box both receive focus. At this point, the cursor is blinking in the text box, waiting for the user to do something.

Each time the user types something, the InteractiveChange event fires. When the user presses the Tab key (that's the sixth InteractiveChange event, by the way), the Valid event fires, and because there is no code in the Valid event to prohibit the user from leaving the control, the first text box loses focus and the second one gets focus.

When the Push Me! button (with a "thisform.release" command in its Click event) is clicked, a number of interesting things happen. First, the button's When event fires, the button gets focus, and the code in the Click event is executed ("thisform.release"). Then the button's Valid event fires, and, interestingly enough, the button's When event fires again! Notice that the Push Me! button's LostFocus event is never triggered.

Finally, the form and then each of the controls is destroyed, the form is unloaded (from memory) and the process is complete. Not exactly what you were expecting, I'll bet. Obviously, it will be well worth your while to spend some time examining and working with the various events until you get comfortable with what fires when, and why.

Now it's time to look at the controls themselves. I'm going to look at each group of controls first, and then examine the specifics of each individual control after that.

## **Visual delineators**

Visual delineators are used to display static information, divide the form, or otherwise provide visual indicators of how the form is organized.

### **Label control**

Use the Label control to place text legends next to text boxes and edit boxes, provide prompts for controls where they can't include the prompt themselves, and provide other textual information such as instructions or reminders. You can change the text of a label and its visible state programmatically while running the form.

The label, although seemingly dumb, has many properties and can react to a number of events. For example, you can click on a label and execute a method, much like an invisible button. You need to be careful with this because good interfaces need to be keyboard-friendly as well, and you can't tab to a label; as a result, there is no "keypress" event associated with a label.

### **Image control**

Use the Image control to place static images on a form. Formats supported include .BMP (bitmaps), .CUR (cursors), .DIB (bitmap), .GIF (Graphic Interchange Format), .ICO (icons), and .JPG (JPEG files).

You can change the image and its attributes programmatically while running the form, and you can control how the image is placed on the form according to the space allotted for it. By setting the Stretch property to 0-Clip, the part of the image that extends past the size of the control will be cut off. By setting the Stretch property to 1-Isometric, the image will be resized so that at least one dimension fits into the control completely. The other dimension will be cut off. In other words, the aspect ratio will be maintained at the expense of part of the image. By setting the Stretch property to 3-Stretch, the image's aspect ratio will be modified so that the picture fits exactly in the space allotted for the control.

### **Line control**

Use the Line control to place lines on a form. Lines are often used to group or divide sets of controls on a form. Dividing a complex form into several areas makes it easier for the user to comprehend the use of the form.

### **Shape control**

Use the Shape control to place boxes and ellipses on a form. Like the Line control, these shapes are often used to group or divide sets of controls on a form. Line and shape controls also can be used to create simple drawings that contain colors, shading, and so on.

### **Page Frame control**

Use the Page Frame control to place multiple sets of controls on a single form instead of creating multiple forms. Each set of controls is placed on a separate page and the pages are placed on top of one another much like a deck of cards. Each page can have a tab that "sticks out" from the deck in a separate location; clicking a tab brings that page to the top of the deck.

Using a page frame is straightforward. After placing a Page Frame control on a form, you'll set the PageCount property to control how many pages are in the page frame. Because much of the allure of a Page Frame control is the nifty set of tabs that display on the top of the control, you'll probably want to keep the Tabs property set to .T., but you can turn the tabs off if you have another mechanism for switching between pages. For example, you can create a wizard interface by placing each step on a separate page of a page frame, setting the Tabs property to .F., and then providing Next and Back buttons for the user to navigate between pages.

Other useful properties include TabStop, which determines whether the user can use the Tab key to move between pages, and ActivePage, which is the number of the topmost page. You can interrogate ActivePage to determine which page is on top, or change its value to programmatically change which page is showing on top.

You can have dozens of pages in a page frame, but you'll see that by default, the tabs are each compressed and the caption on each tab is chopped off. If you need more tabs than can comfortably fit on one line, you can create multiple rows by changing the TabStretch property from 1-Clip to 0-Stack. The Page Frame control will automatically determine the number of rows of tabs necessary so the captions on each page are completely visible. Note that Visual FoxPro automatically creates the rows, so if your tab captions aren't long enough, you can't force multiple rows of tabs.

If you scan through the PEMs of the page frame, you'll see that you can't change the color of the text on each page. However, that's mainly because you're looking in the wrong place. Not only can you change the color of the text, but you can use a different color for each page by manipulating the ForeColor property of a specific page.

It's tempting to create a complex page frame, adding dozens of controls to each page—perhaps even additional page frames on individual pages of the master page frame—but be aware that this won't come cheap. The more complex a page frame is, the longer it will take to load. Be sure to test a form with a page frame against a live data set on a machine typically used by a user of your application. The two-dozen-tab page frame might be snappy on your brand new development machine, but the story might be different for the user.

Probably the most requested feature that hasn't been implemented (as of this writing) is the ability to move the tabs to the side or bottom of the form. But that's why God created ".1" releases.

## **Separator control**

Use the Separator control to place spaces between buttons on a toolbar. Separators can be used to group buttons so the user can more easily identify buttons by noting the group they're in and their relative position within the group. For example, all Edit buttons on the Standard toolbar are grouped together so it's easier to find and use the Cut, Copy, and Paste buttons. Many users don't actually look at the pictures on the buttons, but just click on the "middle button" in order to copy something.

The Separator control will be discussed in more detail in Chapter 14.

## Controls to initiate action

These controls are used to provide the user with a mechanism to initiate an action or to allow Visual FoxPro to initiate an action at a given time. The most-used controls are the Command Button and Command Button Group. Selecting a command button—clicking it or pressing Enter when the button has focus—will cause an event to happen. Examples are bringing forward another form, moving the record pointer, or causing interaction with the database, such as saving data. As with menus, it is good practice to use an ellipsis (a set of three dots) in the caption of a command button when clicking it opens another form.

In stark opposition to Microsoft and other Windows adherents, I believe certain controls should be used only for specific purposes. While it is possible for any control to initiate an action (they all have Click events, for example), I believe it is bad interface design to do so. For example, you often run across applications where a check box opens up a second form. I don't agree with this use. When you click a button, you expect something to happen. When you click a check box, however, you simply expect to have the state of the check box change from checked to unchecked (or vice versa). You don't expect a dialog to display or the form to disappear—well, at least I don't. Others justify this interface, believing that the status of the check box (checked or unchecked) provides a visual clue about whether or not the form opened by that check box contains data.

I prefer to use a command button to open the second dialog, and then modify the caption of the command button to indicate whether or not there is data in the second dialog. For example, with a dialog that has a "Comments" field, I would use a command button titled "Add Comments" when the Comments field is empty. Once the user has entered comments, I'd change the caption to "Edit Comments."

If you choose to use controls for purposes that they aren't intended, be sure to provide plenty of explanation, an escape route should users wander down a path they hadn't meant to explore, and, above all, consistency, so that the same control does the same thing everywhere throughout the application.

## Command Button control

The Command Button control allows the user to initiate an action. It has a full set of properties (such as Enabled, Color, and Caption), and events (including Click and MouseDown).

Most commonly you'll want to attach a code to a command button that is executed when either the control is clicked or the Enter key is pressed when the focus is on the button. This code is placed in the Click event of the button, and it might be as simple as "do FORM XXX" or it might contain a complex procedure or routine, such as posting data to accounts or importing data from another system.

In the scope of the form, there are many other actions you'll want to do with a command button. You might want to change its caption. For instance, you might have two or more mutually exclusive actions to be performed, such as steps in a posting routine. The first step might be to Import new transactions, the second would be to Post, and the third to Archive. Instead of having three command buttons, two of which would be disabled at any given time, you could use one button and simply change the caption (using the Caption property) according to a flag in the table. After reading a record, the following code would change the caption of the command button according to the contents of that flag:

```
do case
case cFlagTrans = "1"
  frmThisForm.cmdTransactions.Caption = "Import"
case cFlagTrans = "2"
  frmThisForm.cmdTransactions.Caption = "Post"
case cFlagTrans = "3"
  frmThisForm.cmdTransactions.Caption = "Archive"
endcase
```

In this case, the button acts as a mini-wizard, seemingly changing state as the process progresses.

In other cases, you might just want to disable a command button, but leave it visible in order to provide a visual clue to the user that the action isn't available or appropriate at a given time. For instance, when moving to a new record, you might want to disable the Save button (using the Enabled property) until the user has made a change. The following code would enable and disable the Save button:

```
if (change has been made)
  frmThisForm.cmdSave.Enabled = .t.
else
  frmThisForm.cmdSave.Enabled = .f.
endif
```

Instead of disabling a command button, you might want to hide it completely using the Visible property. Note that an object can still be manipulated in code when it's not visible.

In keyboard-intensive applications, you might want to provide keyboard hotkeys for command buttons so the users don't have to tab through many controls or move their hands from the keyboard to the mouse in order to execute the button. Preceding one of the characters in the caption with "\<" will make the caption appear with that character underlined. That command button can then be executed by holding down the Alt key and pressing the underlined character.

FoxPro 2.x users should note that the other keyboard hotkeys to mimic Escape and Default behavior (the ! and ? combinations) do not work in Visual FoxPro. Users who are coming from character-based applications are often used to pressing the Escape key to get out of the form. Similarly, they might be used to having a "default" button that is executed upon pressing Enter. The Cancel and Default properties can be set to .T. for a command button. This is handy in order to force the methods associated with that button to be executed at the time the Escape or Enter keys are pressed. For example, you might have code that must be executed upon leaving the form in the method attached to the Done button. By setting the Cancel property of the Done button to .T., that code will be executed whenever the user presses the Escape key to leave the form.

You might also want to place a picture on a command button and possibly change the picture depending on whether the button is depressed or not. The Picture, DownPicture, and DisabledPicture properties enable you to attach an image to a command button to do all this. Note that the image must be appropriately sized for the button.

## Command Button Group control

The Command Button Group control also allows the user to initiate actions, but the buttons are placed in a group as a convenience for the developer. You can manipulate command buttons in a group as a single object, and also tie methods to the entire group instead of to a single command button.

You might want to have a single method called from all of the buttons, and then execute part of that method depending on which button was selected. Attaching the method to the Click event of the command button group will make this happen. You need to interrogate the Value property of the command button group to determine which button was selected. The following code in the Click event of a command button group will execute some common code, then button-specific code, and finally more common code:

```
* button-nonspecific code
<code that is executed regardless of which button was clicked>
* button-specific code
do case
case This.Value = 1
  <code is executed only for button 1>
case This.Value = 2
  <code is executed only for button 2>
caseThis.Value = 3
  <code is executed only for button 3>
endcase
* more button-nonspecific code
<code that is executed regardless of which button was clicked>
```

The command button group is a container, and it is important to note that methods might be attached to similar events in both the command button group and in individual command buttons. The Click method of a single button will execute instead of the Click method of the group if there is code in both.

## Timer control

Use the Timer control to force Visual FoxPro to initiate actions at set intervals. These actions can include proactive actions, such as events that you, the developer, want to initiate, or reactive actions, such as checking the state of something (like the value of a property) at specific times.

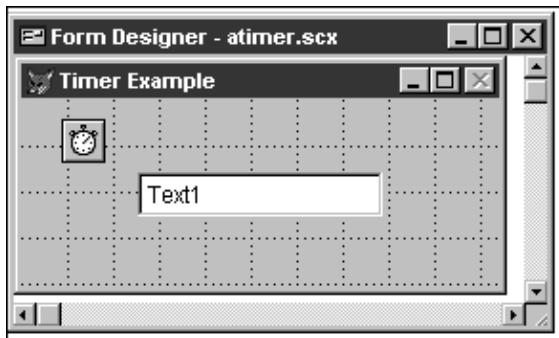
The initial use of a timer can be confusing. Placing a Timer control on a form does nothing by itself. You must set at least one timer property and attach code to one event for the Timer control to do something. The key property is the Interval, which is the time span at which the Timer control fires the timer event. Note that the Interval property is set in milliseconds, so a value of 500 will fire the timer event every half second.

The timer event contains code that is executed each time the interval is reached. Note that the interval doesn't have any direct relation to the system clock. Just because you set the interval to 1000 (one second) doesn't mean that the timer event will be fired at 12:01:00, 12:02:00, 12:03:00. Rather, it will be fired every second, although perhaps in the middle of the second: 12:01:23, 12:02:23, 12:03:23.

Before you get all excited about the capabilities this seems to promise, you should be aware that there are a number of caveats to the timer's functionality. Instead of thinking of a timer as a cure-all to all sorts of problems you've encountered, think of it as providing extensions that didn't used to exist. For example:

- The timer does not process when a menu bar, pull-down menu, or pop-up menu is open. For example, you can cause something to happen when the timer event is fired, such as a WAIT WINDOW command or a message dialog, and the Timer control keeps on ticking. The ATIMER sample in the source code files sets an integer property to 1 and increments it every 100 milliseconds. See **Figure 9.2**. The timer also has a message box that fires when the seconds portion of the current time equals 00, 20, or 40. The following code is in the timer event of the Timer control:

```
thisform.iCounter = thisform.iCounter + 1
thisform.text1.Value = thisform.iCounter
if sec(datetime()) = 0 or sec(datetime()) = 20 or sec(datetime()) = 40
  messagebox("The seconds are divisible by 20." ;
    + chr(13) ;
    + "Wait a few seconds and press OK." ;
    + chr(13) ;
    + "You'll see the timer has kept going." ;
  )
endif
thisform.Refresh()
```



**Figure 9.2.** The *Timer Example* form displays a counter that's incremented every 100 milliseconds.

When this fires, the timer stops processing while the message box is displayed. You'll see that when you eventually get around to clicking OK to get rid of the message box, the counter will continue to be updated with the next value—it hasn't been incremented every tenth of a second while you were waiting.

- The timer has a maximum interval of approximately 3.5 weeks (2,147,483,647 milliseconds). You'll have to perform some tricks behind the scenes to fire an event

once a month. This could be done by setting an interval of two weeks, and then throwing every other timer event away.

- The maximum resolution of a timer interval is constrained by the system clock—approximately 18 ticks per second. So while you can specify an interval of 3 (3/1000 of a second), the interval will actually be 1/18 second—about 0.0555 seconds.
- The interval is not guaranteed to elapse exactly on time. Thus, if you specified an interval of one minute, the timer event might actually fire at 12:01:23, 12:02:24, 12:03:22. (This isn't the fault of VFP; rather, it's because the computer might be doing something else at that specific moment.) Use the system clock in combination with the timer in order to manage precision.
- An application demanding heavy processor resources can cause timer events to be processed late—or not at all. Consider this when deciding how short an interval to rely on.

## **Hyperlink control**

The Hyperlink control allows you to let a user jump to a specified URL. Its navigational capabilities also include moving back and forth through a container's history, much like a browser's Back button allows you to move to the previous Web page.

The Hyperlink control is somewhat of a disappointment when compared to the other native controls in Visual FoxPro, for two reasons. First, you can't really use it "out of the box" like you can the other 20 controls. It has no user interface and requires a fair amount of work to implement robustly—much more than, say, dropping a check box on a form and then being able to click on it. Second, the Hyperlink object works only with Microsoft Internet Explorer. This is corporate arrogance at its worst. Imagine if the Text Box control worked only on computers that were attached to a Windows NT network—if your application was running on Novell or LANtastic, you'd have to use a different, more complex mechanism!

## **Controls to manipulate data**

Data-manipulation controls can be divided into two primary groups: those used when the data to be entered can't be predetermined, such as a company name or a phone number; and those used when the user will make a selection from a predetermined list of choices, such as the name of a department, the sex of an individual, or the status of a payment. Text Box, Edit Box, and Combo Box controls allow the user to enter information that can't be predetermined, while the Option Group, Check Box, Combo Box, List Box, and Spinner controls can restrict the choices available to a user. The Grid control is a spreadsheet-like object that can contain both types of data controls.

### **Text Box control**

Use the Text Box control to display and edit data of a fixed length. This control is usually tied to a non-memo field in a table but can also be used to enter data that the application can use elsewhere, such as the heading of a report or the name of a file being created. Use a text box

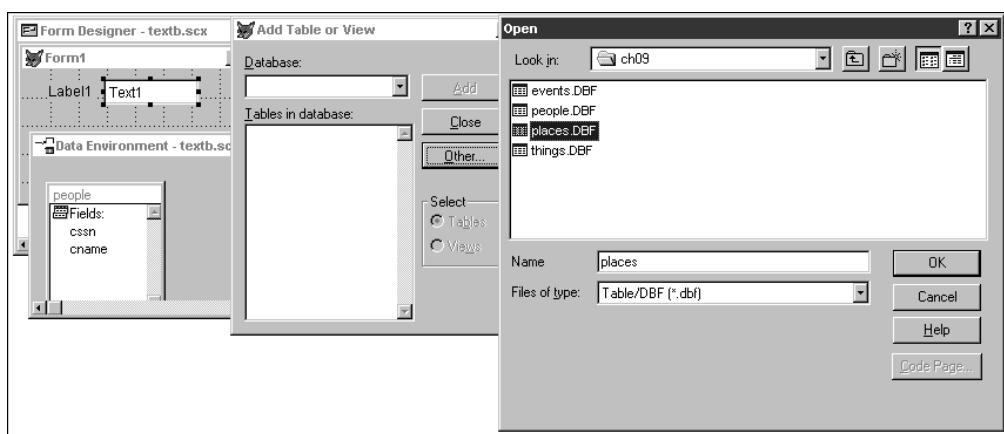
when the value that a user will enter can't be predetermined but the length of the data will fit in a non-memo field.

To create a Text Box control that will not be bound to a table, select the Text Box control from the Form Controls toolbar and drag it to the form. Resize the control as desired, give it a name, and you're ready to use it. Once you have a Text Box control on a form, you're likely to want to type data in the text box and then determine what data has been typed there. Typing data into the text box is simply a matter of moving to the control and typing. Once done, you can move out of the text box. The data that has been typed in the text box can be determined by interrogating the Value property of the control.

To bind a Text Box control to a table, set up a data environment that contains the table or tables desired, and then drag the specific fields to the form. (If you attempt to drag an entire table to a form, you'll automatically create a Grid control, which is addressed later in this chapter.)

If you created a Text Box control by dragging a field from a data environment onto the form, you'll see that the ControlSource property for the text box has automatically been assigned the value of the table and field name. This makes life very easy. To create a simple navigation screen, place the usual Next and Previous command buttons on the form. Then, when running the form, selecting the navigation buttons will automatically cause the text boxes to display data from the table. The ControlSource property causes the appropriate table(s) to be opened and data displayed when the form is run.

If you need to bind a Text Box control to a table but, for whatever reason, don't have the ability to drag fields from the data environment, you can still avoid typing the data source manually. First, add the table to the data environment. You can do this by right-clicking on the form, selecting the Data Environment menu command, and when the Open dialog appears, select the table of interest. After clicking OK in the Open dialog, the table will be added to the data environment. (You'll have a chance to add more tables to the data environment by using the Add Table or View dialog, as shown in **Figure 9.3**.)



**Figure 9.3.** Clicking the Other button in the Add Table or View dialog allows you to add additional tables not in a database to a data environment.

Once the table in question is part of the data environment, select the ControlSource property of the Text Box control in the Properties window. A combo box appears, which enables you to either type a data source or select a field from any open table. Scroll through the list of fields and select the desired field.

Because the text box is often the primary, if not only, tool for data entry into many applications, you're typically going to require many demands of the Text Box control. First, you're probably going to want to format or mask the input of the data entered into a text box. For example, when entering a Social Security number, you might want to store only the numbers, but display the number with the hyphens. You can do this by using a format of R and an input mask of 999-99-9999. The Help topic "Template Codes" has a complete list of format codes for text boxes. You'll often want to validate the data entered by the user. This is most commonly done by using a method in the Valid event of the control. This method must return .T. or .F. depending on whether or not the value is allowed. For example, if you were validating a birth date, you'd place the following code in the Valid event:

```
txtDOB TextBox:  
if This.Value > date()  
    return .f.  
endif
```

Note that you'd probably want to include some mechanism—perhaps a Wait Window message or a Message Box—that explains to the user why the entry was not valid. This will get you started, but I'll show you a better way to handle data validation in text boxes as well as other controls in Chapter 10.

It is often handy to disable a text box when you don't want the user to edit the contents, and you might want to provide a visual clue to the user when you do so. Use the DisabledBackColor and DisabledForeColor properties. Be sure to select color choices that are visible to the user; it's easy to mistakenly select two colors without enough contrast.

As another visual aid, you might want to mark the data in a text box as read-only. You can do this by disabling the field, of course, but that could confuse the user into thinking that there is some situation where the data will not be read-only. Better to use a different visual indication that the data will never be able to be edited. You can do this by setting the ReadOnly property to .T.

Depending on the type of data entry being done in the form, you might want to automatically select the contents of the entire field when the text box receives focus. Setting the SelectOnEntry property to .T. will cause the contents of the field to be selected when the user tabs into it.

Alternatively, you might want to automatically move the cursor to the beginning of the field when the text box receives focus. Placing the following code in the GotFocus event of a text box will do so:

```
keyboard "{home}"
```

---

You might often find that you want the user to be able to decide whether these capabilities should be implemented in specific locations. Instead of hard-coding the code in each method, consider the following method:

```
if m.lUserGoHome  
    keyboard "{home}"  
endif
```

In another part of the application, you would allow the user to set a flag that assigns a .T. or .F. to the memory variable m.lUserGoHome so they can choose whether or not the option would be implemented.

One of the most frequently asked questions is how to inhibit the display of passwords or other sensitive data. In previous versions of FoxPro, developers had to resort to any number of tricks, including changing the font of the password field or using a third-party utility that would trap the keystrokes and return a “dummy” character to the screen while actually processing the character that was typed. The PasswordChar property allows you to specify a character that will be displayed in the text box in lieu of the actual entry. Note that the data typed by the user will still be treated as normal—it’s simply the display of the data that is modified.

## Edit Box control

Use the Edit Box control to display and edit data of an unlimited length. This control is usually tied to a memo field in a table. It essentially functions the same as a text box but has some additional properties. The ScrollBars property can be set to 0-None or 2-Vertical. The AllowTabs property can be set to .T. or .F. depending on whether you want the user to be able to use the Tab key inside the memo field. You can also set the ReadOnly property to allow users to scroll through the memo field without being able to edit the contents of the field.

## Check Box control

The Check Box control is used for two disparate purposes. The first is just like the other data controls where you’re moving data in and out of the tables: The field can contain one of two elements, such as Male/Female, or Is Alive/Is Not Alive. Instead of making the user enter a “M” or an “F”, you could use a single check box. However, you need to be careful when you use controls for this. For example, you might be thinking Male/Female is exclusive, but you also might need to store a third choice: Unknown.

The other choice is simply to control the form. Say you’re going to send something to the printer but first you want to preview it on the screen. You can do this by using a check box called Preview. This way it has nothing to do with the table or data.

You can control the status of the check box by initializing it to true; otherwise the default will be false—empty or blank. And, like other controls, you can change the value of the check box by changing its Value property.

You can change the caption of a check box by modifying its Caption property.

You can read and modify the value of a check box in one of two ways. First, a check box can have a numeric value. 0 means the check box is unchecked, 1 means it’s checked, and 2 means it’s grayed out. This does not mean it is disabled—it means we don’t know what the value is. Typically, you’d use a value of 2 when the field is a null. (Remember, null doesn’t

mean empty; it means “I don’t have a value for this.”) If you used a field to record whether an individual has seen Elvis, a gray box would mean you don’t know if that person has seen Elvis.

The second type of value a check box can have is .T., .F., or .null. These represent a checked box, unchecked box, or grayed-out box, respectively.

You can also disable a check box, just as with other controls, by setting the Enabled property to .F. Note that a check box has a DisabledForeColor that by default is dark gray—if your form’s background color is light gray, the white will stand out like a sore thumb. Be careful with the color you use so that you don’t make a disabled check box unreadable.

## **Option Group control**

Use the Option Group control to present a fixed number of choices to the user, only one of which can be selected at a time. These used to be called Radio Buttons, modeled after the buttons on a car radio, which had the characteristic that only one button could be selected at any one time. Technically, you can also leave all of the option buttons in a group unselected, much like when you hadn’t selected any station on the radio. I prefer to include a choice in the option group to indicate that none of the other choices were acceptable. For example, instead of having two choices for gender—Male and Female—and then leaving both option buttons unselected if the gender was unknown, I’d include a third for Unknown. The reason is that it’s impossible to physically uncheck all of the option buttons. (On a car radio, you could unselect all of the buttons by pressing two in at the same time. You’d need a pretty big mouse to do that on your computer screen. Seriously, you can.) If you accidentally selected one of the option buttons and later wanted to reverse that decision, there is no way to do so if you didn’t give yourself an out.

You can use an Option Group control to let the user control additional choices on the form, and you can use it to move data in and out of a table. Some properties vary according to the way you use an option group.

An option group is actually a container for another control: the Option Button. When you address all of the option buttons as a group, you need only address the option group container object. When you address a single option button, you need to address both the container and the option button.

Much like the Command Button control, you can determine (and change) the caption of a specific option button by using the Caption property. Note that you need to know the name of the specific option button. You can determine which one is selected with the option group’s Value property. Notice that this is a property of the option group—not a specific option button. You can then determine (and change) the caption of the currently selected option button in a group by using the opg.Buttons(opg.Value).Caption property.

Often you’ll want to use an option group to move data between the form and the table. Because these values are like “lookup” values, you might think you’d use a lookup table with codes, and store the code of the option button in the actual data table. However, because an option group indicates that the given choices are expected to be fairly (or, actually, extremely) stable, it is probably a safe bet to store the chosen option button’s caption directly in the table. You might want to alter this strategy if you expect to change the captions based on other conditions in the form.

You might also want to store the number instead of the caption if you are constrained by size (the caption will require a longer field than the number) or if you will be translating the caption to a different language for international use.

To bind the option group to a table, create the form, create a data environment, create the option group, and select as the ControlSource the name of the field in the table that matches up with the option group. Note that the ControlSource property belongs to the option group—not to a single option button—because the option buttons are simply various choices that might be placed in the field. The option group is tied to the field in the table. Just be sure that the values in the table match the captions on the buttons exactly. This is actually only an issue if the Field and Value of the option group are both of Character data type. The values in the table should range from 1 to the value of the option group's opgButtonCount property if the value is numeric.

## List Box and Combo Box controls

Use the List Box control to present a predetermined set of choices to the user. Use the Combo Box control to display a list of predetermined choices but also allow the user to add choices not already provided. This “combination” of functionality gives the control its name. List boxes and combo boxes have much in common, so I’ll cover those features just once for both, and then I’ll discuss unique features of each control.

There are three variations on the list box and combo box type of control: one of each and a third that is sort of half-list and half-combo. The regular list box displays a number of choices in a rectangular box and has a scroll bar on the right that you can use to view additional choices that don’t fit in the available space. The combo box allows users to either pick from a list of existing choices by clicking the down arrow button or to enter a new choice as they would in a text box.

The third type—a drop-down list box—provides the same capability as a list box (selecting from a set of predetermined choices), but it takes up less screen real estate because it is presented to the user as a combo box control—a single pop-up control in addition to a down-arrow button that is used to expand the box.

I’ll treat both the regular and drop-down list boxes the same because the combo box has additional functionality not found in either list box.

### Uses of lists and combos

First of all, I should discuss the proper uses of list and combo boxes, because there’s more to these controls than meets the eye.

You can use a list box or a combo box to allow the user to select one item from a predetermined list. One common example is a pick list, where a list box displays the choices from which the user can select an item. See **Figure 9.4**, where the Add Printer dialog uses a list to display all the printers. Note that the printers that have already been selected have been disabled so they can’t be selected again.

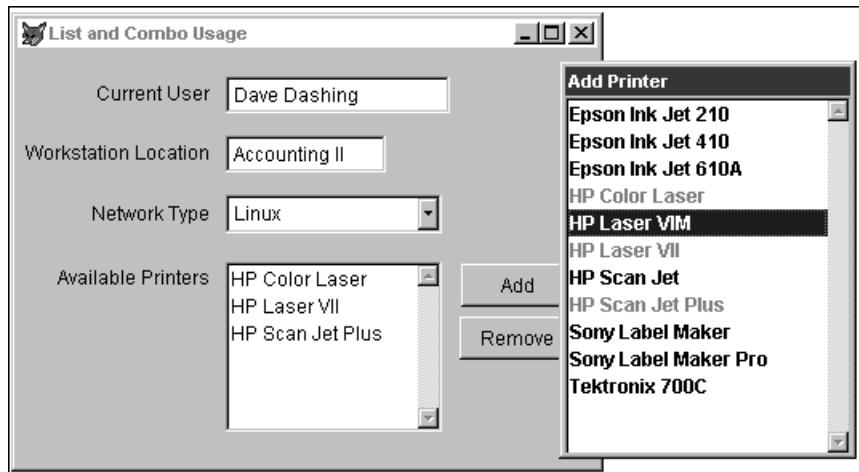
Another example is the ubiquitous Open File dialog, where a list box presents a list of folders through which the user can navigate.

A combo box, on the other hand, not only can allow a user to choose from a list of predetermined choices, but also can easily show which choice from that list has been selected.

You can also use a list box to display multiple values that are related to another entity, particularly when the number of values will vary. For example, you might have a form that displays all the information for a family, and you would use a list box to display the names and ages of all of the kids. Normally, you’ll need to provide a mechanism for allowing the user to

add items to the list, remove items from the list, and edit items that were already in the list. In Figure 9.4, I've used a pair of buttons, Add and Remove, to provide this functionality. If the user needed to edit an item, double-clicking it would open a dialog that would allow the user to do this.

Note that you can also use a combo box for this purpose. You could populate the combo box with all of the values, and display only one of them. However, you'd only want to do so if screen real estate was extremely limited—and even then, you'd want to provide some sort of visual clue that there was more than one item in the combo box.



**Figure 9.4.** This form uses a combo box to display the type of network and a list box to show which printers are attached. A second list box allows the user to add more printers as desired.

### Populating a list or combo box

The very first thing you need to do when using a list box or combo box is to populate it. You can fill a list box in one of 10 ways, and each method is fraught with its own peculiarities. In each case, you set the RowSourceType of the control to a specific option and then set additional properties according to which RowSourceType you selected. In general, the trick is that the data that will populate the control must be visible outside the scope of the control itself. In other words, if you use an array to populate the control, you can't create the array inside a method that belongs to the control because the array won't be visible outside that method. The following sections describe the available methods of populating a list box or combo box.

### ***Programmatically***

The first method is to not populate the list box at all. In this case, you would leave the list box empty and then programmatically fill it later. You set the RowSourceType property to 0-None, and then, when needed, use the AddItem method to add items to the list box. Here's the code that populated the list of networks in the combo box in Figure 9.4:

```
thisform.combo1.AddItem("Novell")
thisform.combo1.AddItem("Windows NT")
thisform.combo1.AddItem("Linux")
```

Precede the value you are adding with a backslash if you want to disable that value. For example, here's the code that populated the Add Printer list box:

```
thisform.list1.AddItem("Epson Ink Jet 210")
thisform.list1.AddItem("Epson Ink Jet 410")
thisform.list1.AddItem("Epson Ink Jet 610A")
thisform.list1.AddItem("\\HP Color Laser")
thisform.list1.AddItem("HP Laser VIM")
thisform.list1.AddItem("\\HP Laser VII")
thisform.list1.AddItem("HP Scan Jet")
thisform.list1.AddItem("\\HP Scan Jet Plus")
thisform.list1.AddItem("Sony Label Maker")
thisform.list1.AddItem("Sony Label Maker Pro")
thisform.list1.AddItem("Tektronix 700C")
thisform.list1.DisplayValue = "HP Laser VIM"
```

### ***Hard-coded values***

The second method is to hard-code values for the list box. While this method is easier than other methods that require setting up arrays, filling cursors, or attaching tables, it obviously has the huge downside of increased maintenance. You set the RowSourceType property to 1-Value, and then type the actual data items in the RowSource property—yes, directly into the Property window. One small trick: Many of us are used to leaving a space between items in a comma-delimited list. Visual FoxPro interprets all of the characters between the commas as part of the item, so a space between a comma and the next item will cause the next item in the list to be indented.

### ***First N field values from a table***

The third method is to display fields directly from a table. Note that you can choose to select the first N fields, where N is any number you wish, but multiple fields will appear in the order that they appear in the table. If you create your tables like I do, using a surrogate primary key as the first field, this choice won't be terribly useful. If you want to control the order in which the fields display, or if you want to display fields from multiple tables, use the SQL Statement method, discussed next.

You set the RowSourceType property to 2-Alias, set the RowSource property to the name of the table that will populate the list box, and then set the ColumnCount property to the number of columns to be displayed. It's important to note that as you move through the list box, you are actually moving the record pointer through the table.

### **SQL Select**

The fourth method is to create a custom set of fields via an SQL Select statement. You can choose the order of the columns, the order of the items (via the ORDER BY clause in the SQL statement), and even pull data from more than one table if you like.

Set the RowSourceType property to 3-SQL Statement and type the SQL statement (yes, the whole thing!) into the RowSource property. Note that you have to be careful about the syntax of the SQL Select command. For instance, you have to make sure that variables you reference in WHERE clauses are visible to this control. You can't initialize the variable in a method elsewhere attached to this control. Also, be sure you explicitly send the results of the SQL Select statement to a cursor or a table—if you don't, you might see an unintended Browse window displaying the results of the query.

Don't forget to consider multi-user requirements at this stage. It's easy to send the results of one of these SQL Select statements to a hard-coded name for a table, and then run into problems with multiple users. Consider using SYS(3) or another mechanism to create unique file names or use a cursor. You'll need to prepend a character to a table name created by SYS(3).

Because the SQL statement is a character string, you'll need to surround it in quotes if you want to create or modify it programmatically:

```
thisform.list1.RowSource = "select Name, City from FRIENDS into table 'Z' + sys(3)"
```

You'll want to make sure you close the cursor or table manually in the Destroy method of the control.

### **Query**

The fifth method is to populate the list box with the results of a query file (MYQUERY.QPR) designed and saved in the Query Designer. This will produce similar results to the previous method except that you don't have to type the entire statement in the RowSource property.

Set the RowSourceType property to 4-Query and then enter the name of the query file into the RowSource property. Visual FoxPro will assume an extension of .QPR if you don't include one.

### **Array**

The sixth method is to populate the list box with the contents of an array. This is powerful because you can use a two-dimensional array to create a multicolumn list box, and it's extremely fast because you're working only with memory. Note that if you create an array by selecting items from data tables, you can potentially create very large arrays that might exceed the available resources on the computer.

There are two other issues you might want to consider. First of all, you might encounter performance degradation as the size of the array increases significantly. A rule of thumb is to keep the number of elements in an array under 500, but the number will depend specifically on the performance of the machine and resources available. The second issue is that a list or combo with hundreds or thousands of items might be difficult for a user to navigate. If you've got several thousand items from which the user can select, it's possible that the user might have

---

a difficult time distinguishing between similar items. You can circumvent this by including enough information to make each item unique, or by including multiple columns in the list.

You can use an array created elsewhere in your application or address a custom array property of the form or formset. You have to be sure that the array is scoped properly in relation to the control. In other words, trying to declare an array in a method that's on the same level as the List Box control will cause the array to disappear when leaving the method. Note that you can still populate and repopulate the array from within any method as long as the array has been declared and initialized properly. The following code, run outside of a form, will declare and initialize an array that can then be populated via an SQL statement inside the form:

```
declare aSomething[1]
aSomething = ""
```

The optimal method is to create an array property of the list box or combo box itself, and I'll show you how to do that in Chapter 10.

In both cases, you can display multiple columns in the list box from the array by setting the list box's ColumnCount property to the desired number of columns. Note that you might have to set the Column Width property of the list box in order to make its columns appear as you need, because, unlike fields, Visual FoxPro has no way of telling how wide the contents will be.

All in all, however, arrays are very powerful. You'll see in Chapter 10 that they are my row source of choice.

### ***Delimited field list values from a table***

The seventh method is to populate the list box with a comma-delimited list of fields. This provides the same functionality as RowSourceType = 2-Table with the additional benefit of being able to specify which fields and in which order. RowSourceType = 2-Table only allowed the first "N" fields. You are, however, still limited to fields from one table. Be sure to specify the alias in the expression for only the first field—not every field—like so: customer.company, city, state, postalcode, as opposed to customer.company, customer.city, customer.state, customer.postalcode.

### ***File list***

The eighth method is to populate the list box with the names of files. The list box is by default populated with all files in the current directory, but you can specify a file skeleton to show only certain file names. You can also change the drive and directory. Set the RowSourceType to 7-Files and the RowSource to the file-name skeleton (for example, \*.txt).

### ***Field list***

The ninth method is to populate the list box with the names of fields from a table. Set the RowSourceType property to 9-Structure and specify the name of the table in the RowSource property.

### **Pop-up menu**

The tenth method is to populate the list box with the contents of a previously defined pop-up menu. This option is included for backward compatibility where you would be transporting existing code into the Visual FoxPro environment and have pop-up menus already defined. This choice displays actual field names, not user-friendly captions, so it's probably not going to be suitable for your users.

### **List Box functionality**

The List Box control has a wide range of features that make it the control of choice in many situations. A list box by nature has incremental search capability—when the control has focus, typing the first few characters will move the focus highlight down the list to the choice that most nearly matches the typed characters. You can't tab over to other columns in the list and do incremental search on those columns. They are there just for additional description—for example, a customer name and account number, a person's last name and first name, or a company name and the name of the city. If you like, you can turn off this incremental search functionality. Set the IncrementalSearch property to .T. or .F. as desired.

A nice feature of Visual FoxPro's List Box control, as opposed to earlier versions, is that it can include multiple columns, use a proportional font for the display, and still have the items appear in neat columns. In earlier versions, we were required to use an ugly nonproportional font in order to make additional columns line up. You'll need to set the ColumnWidths property in order to show the right amount of data in each column.

Additionally, you can include mover bars in a list box so that the user can rearrange the order of the items in the list (set the MoverBars property to .T.), and allow the user to make multiple selections in the list (set the MultiSelect property to .T.). One frustrating aspect of mover bars, unfortunately, is that they are available only when the RowSourceType is 0 or 1.

### **Combo box functionality**

The combo box has an additional property that allows you to add a value that the user types into the items in the list. The DisplayValue property returns the value of the item the user has typed. Use it in combination with the AddItem method to add the value to the list. You'll probably want to institute a check to make sure you're not adding duplicate items. The following code in the Valid event will perform both of these functions:

```
if not This.Value == This.DisplayValue
    This.AddItem(This.DisplayValue)
endif
```

### Defining and returning values from a list box

You can determine which item has the highlight by the `DisplayValue` property. For example, in Figure 9.4, the Laser VIM printer was highlighted by using the following line of code (note that the value “HP Laser VIM” had already been added to the list):

```
thisform.list1.DisplayValue = "HP Laser VIM"
```

You can determine which item in a list box is currently selected with the `Value` property. For example, the command:

```
thisform.txtCurListItem.Value = thisform.lstDest.Value
```

will place a copy of the current list box selection (`lstDest.value`) into the text box named `txtCurListItem`.

If you’ve set the `MultiSelect` property to `.T.`, you’ll need to process those selected. The basic strategy is to loop through the items in the list and determine whether or not a specific item has been selected by interrogating the `Selected` property. Once you’ve made that determination, you can use the `List` property to identify the value of the selected item (the `Value` property will identify only the last selected item in the list) and do with it what you will.

One mechanism that takes advantage of the `MultiSelect` property is a “mover” control, in which you display two lists of objects and allow the user to move more than one object to the other list at one time.

### Spinner control

A Spinner control consists of a text box and two arrows. It is used to allow the user to enter a value or increment the value visually by clicking the arrows. Clicking the arrows will force the displayed value to “spin” through values like the numbers on a gas pump display.

You can bind the control to a field in a table by using the `ControlSource` property, just as with other controls.

The Spinner control has several special properties, including the ability to set the value of the increment as well as the maximum and minimum values allowed by spinner and keyboard entry. In other words, you can force the value to increment by 5 or -5 when one of the arrows is clicked, and to stop incrementing when a value such as 100 or 2048 is reached.

You can use a spinner to increment non-numeric values, including dates and character fields. First, narrow the width of the spinner so that only the up and down arrows appear. Next, create a text box, named `txtDate`, and initialize the value as today’s date with the following code in the `Init` event:

```
this.Value = date()
```

Finally, update the value in the `txtDate` text box when the up and down arrows are clicked by including the following code in the `UpClick` and `DownClick` events, respectively:

```
UpClick: thisform.txtDate.Value = thisform.txtDate.Value + 1  
DownClick: thisform.txtDate.Value = thisform.txtDate.Value - 1
```

It's just slightly more involved to make a character string increase or decrease incrementally with a spinner. The trick is based on how you want to define the increment of a character. Suppose you want to just roll through the alphabet, A to B to C, and stop at the letter Z. Following the same process as with the date spinner above, create a spinner where only the arrows show and a text box, txtAlpha, that is initialized to the letter "A". The following algorithm will return the next character in the alphabet:

```
char(asc( txtAlpha )+1)
```

You can place a version of this algorithm in the UpClick and DownClick events of the spinner. Note that you might want to do some bounds checking so you don't end up trying to increment past the range of ASCII characters.

```
* UpClick()

if thisform.txtAlpha.Value = "Z"
  m.nX = messagebox(You can't increment past 'Z', 32, "Help me...")
else
  thisform.txtAlpha.Value = chr(asc( thisform.txtAlpha.Value )+1)
endif

* DownClick()

if thisform.txtAlpha.Value = "A"
  m.nX = messagebox(You can't decrement below 'A', 32, "Help me...")
else
  thisform.txtAlpha.Value = chr(asc( thisform.txtAlpha.Value )-1)
endif
```

A common mistake with spinners is accidentally setting the increment value to 0. The spinner will appear to be broken when the problem is that the value is incremented by zero each time it is clicked.

There is a caveat to using increment values other than 1 or maximum and minimum values. If the increment value doesn't divide equally into the maximum and minimum values, the increment will be ignored when you hit the limit value. For example, suppose you've set an increment of 5 and a maximum value of 23. Clicking the up arrow will change the spinner value to 5, 10, 15, 20, and then, finally, 23, which might not be what you want.

## Grid control

The Grid control is a spreadsheet-like control that can be used to present tabular data. The grid, like the command button group, option group and page frame, is a container control. Like the option group, a grid by itself contains no data. Instead, it holds one or more columns, each of which is also a container. A column holds a header (the title for the column) and one or more controls. This control defaults to a text box, but you can replace the text box with a different type of control, such as a spinner or an edit box, or you can even place multiple controls in a column, any one of which is displayed at a given time.

The grid can be viewed as an alternative to a list box, and, indeed, is a much more flexible control. You'll immediately gravitate to the grid in three specific instances. The first is to display the "many" side of a parent-child relationship or a master-detail relation. Examples

include all of the invoices for a customer, and all of the line items for a purchase order. The second instance is to take over for a list box when the list can't handle the number of rows, due to memory or other resource constraints. You might also want to use a grid when the values you want to display have so many attributes (columns) that you need the ability of a grid to be able to scroll left and right. (The list box is still the preferred choice when you need to populate the control from an array or when you need to provide multi-select capability—neither of which can be done with a grid except with a lot of work.)

Each component of the grid has a number of properties, and, like other container controls, is addressed through the control hierarchy. For example, the grid and each column all have width properties. They are addressed like so:

```
form1.grid1.Width = 250  
form1.grid1.column1.Width = 75  
form1.grid1.column2.Width = 125
```

It would take the better part of this book to explore all of the properties and methods of each component of a grid, but it's worth a few moments to mention some of the more useful ones. You can control the position, size, and colors of all components as well as font attributes like FontName, FontSize, FontBold, and so on. You can control the way the entire grid looks by manipulating the DeleteMark, RecordMark, ScrollBars, and GridLines properties, and whether the user can move or resize columns or edit the contents with the Movable, Resizable, and ReadOnly properties. The ColumnCount property determines how many columns appear in the grid.

Besides addressing the property of a single control, you can use the SetAll property to manipulate the same property for multiple objects. For example, instead of typing:

```
form1.grid1.column1.header1.FontSize = 12  
form1.grid1.column2.header1.FontSize = 12  
form1.grid1.column3.header1.FontSize = 12
```

you could use the single command:

```
form1.grid1.SetAll("FontSize", 12, "header")
```

You can bind data to a grid in one of two ways. You can drag a table from the data environment (use the title bar of the table window in the data environment) to the form and the grid will be created automatically. The downside is that you don't have explicit control over the controls inside the grid, because it is populated with the columns from the table starting with the first field.

The second way is to create an empty grid and manually add columns. First, set the ColumnCount property of the grid to the number of columns you want. Then, set the ControlSource property for each column. If you have a table in the data environment, the drop-down list box in the Properties window for the ControlSource property will be populated with the fields from the table.

If you have a parent-child relation set up in the data environment, dragging fields from the parent table and then creating a grid from the child table will automatically set up a one-to-

many relationship in the form. Be sure to set the InitialSelectedAlias of the Data Environment to the parent table.

You are not limited to using a text box as the control in a column. However, adding a different control is a bit tricky. Here's how to do it.

1. Create a grid on a form.
2. Select the column to which you want to add the control (let's say it's an edit box and we're adding it to a column that maps to a memo field in the table). Remember that you can use the Object drop-down list box in the Properties window to select the column of interest.
3. Select the Edit Box control from the Form Controls toolbar by clicking the Edit Box icon. Select the Form Designer again—by clicking the title bar of the window—and then click in the desired column. You'll see the form flash for a second and then return to normal. Look at the Object drop-down list box, and you'll see that a second control, Edit1, has been added to the column container below Text1.
4. Here's the tricky part. If you simply run the form at this point, you'll see the memo field appear as a regular memo field with the word "memo" in that column, just as it does in a regular Browse window. This isn't exactly what we had in mind. We need to specify that we want to use the edit box as the control for this column.

Go back to the column container and change the CurrentControl property for the column that contains the additional control. You'll see that you can use the drop-down list box immediately above the page frame in the Properties window to choose between the original Text1 control and the new Edit1 control. If you like, you can select the original Text1 control and delete it from the grid.

5. Run the form. When you move into the column containing the memo field, you'll see that the control changes to an edit box, complete with a scroll bar if necessary. When you move out of the field, the control changes back to a regular text box. If you want to see all memo fields, you can change the Sparse property of the column to .F. Generally, you'll want to keep Sparse set to .T. for better performance.

## Other controls

### Container

After you've worked with some of the native containers in Visual FoxPro, such as forms, option groups, and page frames, you might be thinking that it would be nice to be able to create your own containers, and fill them with a set of controls that you'd like to use over and over again. This functionality, in essence, is what object-oriented programming is all about—creating your own objects and then reusing them.

The Container control is often used as part of this process of creating your own objects, but it doesn't have a lot of use on its own. For this reason, I'll defer in-depth discussion until Chapter 10.

## ActiveX control and ActiveX bound control

These don't fit into one category because, by definition, they could be anything. I'll cover ActiveX controls in depth in Chapter 23.

## Custom class

Not all classes in Visual FoxPro are used to create visible controls. The Timer is one example of a control that doesn't ever appear to the user. Another class, the Custom class, is similar to the Container class in that it allows you to create your own class with properties, events and methods, but it's different because it has no visual representation. Thus, it's useful for creating class libraries that handle behavior or functionality that you would have used a procedure file for in the past.

## Session class

Another non-visual class, the Session class is new to Visual FoxPro Service Pack 3. It allows you to create a custom, user-defined object that manages its own data session. Before the Session class was available, if you needed to work with tables or views that required a data environment, you had to create a form, add a data environment to the form, and do your work there. But you also had to set the form's Visible property to .F. This by itself wasn't a huge hurdle, but you also had to deal with the fact that a form class had a lot of overhead associated with it—a lot of properties, events, and methods that had to do with its visibility, and, thus, were useful in terms of a non-visual class.

You won't find information on the Session class in the regular Visual Studio MSDN Library help. However, there is a help file in the Microsoft Visual Studio\VFP98 directory called VFPSP3.CHM that lists some additional information such as PEMs to get you started.



# Chapter 10

## Using VFP's Object-Oriented Tools

You may be tempted to skip this chapter. You've already learned how to build a menu, create some forms, and drop controls on those forms. You might be thinking, "Skip to the Report Writer section, and I'll know all I need to start building Visual FoxPro applications." Actually, that's not exactly true. You'll have all you need to know to build FoxPro 2.x applications.

Next to SQL, the most wonderful feature of Visual FoxPro is its implementation of object-oriented programming. To skip this chapter would be one of the biggest mistakes of your programming life. In this chapter I'm going to show you what object-oriented programming is, with examples that will make sense to you, a database programmer; how VFP's object-oriented tools work; and how to incorporate them into the forms that you've already learned how to build. Finally, I'll set you up with your own set of base classes that you can use to start building real-world applications—today.

A lumberjack wandered into town after being in the mountains for years and years. First thing on his list was a bath; after that, he stopped in at the general store and asked for a new ax, since the one he had was 20 years old and was getting just plain worn out. The owner asked him if he wouldn't prefer a chainsaw. The lumberjack gave him a quizzical look but reckoned that he'd like to take a look. There it was, with a bright red cover, a shiny steel handle, and hundreds of razor-sharp teeth on the chain. The lumberjack, big as he was, hefted the device in a single hand and swung it around like it was a hand ax. The store owner promised he would cut down 10 trees a day with this thing.

"Ten trees? On my best day I've never cut down more than two."

"Ten trees a day or you can have it for free," pledged the store owner.

"Well, that sounds pretty good," answered the lumberjack. "Let me have it." As he started to walk out the door, the owner stopped him, saying, "Let me show you how to use it." The lumberjack pushed him away, saying, "I think I kin figger this out myself."

A week later the lumberjack staggered back into town, shivering and shaking, clothes torn, blood on his hands and drool hanging from his chin, swearing at the store owner. "I worked my tail off this week and I could never get more than three trees down a day. It's a helluva lot better than my ax, but there's no way a man could cut 10 trees in a day."

"Only three trees?" asked the storeowner. "Hell, my grandmother could cut down 10 trees a day with that baby. Give it here and let me see what's wrong." He started it up and the lumberjack jumped back, alarmed. "What's that noise?"

So far in this book, we've seen the incredible power and flexibility of Visual FoxPro—we can create blindingly fast applications with visually appealing, sophisticated interfaces and user-friendly features galore. I'm sure that more than a few of you are already overwhelmed with the number of tools, dialog boxes, objects, choices, commands, and other minutiae to

remember. The last thing you need to hear is that we have a whole additional level of functionality to learn. However, it's true. We've skipped one feature of VFP almost completely. And it's this capability that will increase your productivity to unheard-of levels.

I'm talking about Visual FoxPro's object-oriented programming capabilities. You can use Visual FoxPro without taking advantage of its object-orientation capabilities, but it's like using a chainsaw without starting it up.

The industrial revolution started when people stopped making items one at a time and used the concept of interchangeable parts to create a large quantity of the same widget. Productivity went up, quality improved, and economies of scale were realized when the manufacturer could crank out 20 of the same item instead of custom building each one. The resulting cost savings meant a greater number of buyers, which meant increased profits, some of which were reinvested in improving the tools. However, many people don't realize that the concept of interchangeable parts wasn't invented in 1820; it had been around for centuries. But it took the ability to smelt metal to precision tolerances so a manufacturer had the ability to reliably reproduce a part to specific measurements. The concept was there but the environment wasn't ready.

The same is true for us in the software industry. The mechanism of creating interchangeable parts in software is object-oriented programming (OOP for short). Instead of writing the same code over and over, we create the mold once and then reuse it over and over for multiple applications. Of course, we've been hearing about OOP for years and years—almost as long as we've been hearing about artificial intelligence. But the promise and the delivery have been two different things. Why?

As with the development of manufacturing during the industrial revolution, the environment hasn't been ready. Now, many of you have the power of a small Cray sitting under your desk in the form of a 256 MB, 10 GB Pentium III with a 21-inch flat-screen monitor and a color graphical user interface. True object-oriented programming requires a lot of horsepower. We now have it.

The basic idea behind the promise of object-oriented programming is that we should be able to create an object—a form, a control on a form, even a function that doesn't have a visual component. Instead of cutting and pasting a copy of that object, we'll make references—pointers—to that object in the actual application. When the object changes, the application's objects change, too. Automagically, as they say.

VFP has this capability. Those who use this capability will be cooking with gas grills, while the rest of the crowd will still be sticking raw meat on a sharp stick into a campfire. The tools that enable us to do OOP are the Class Designer and the Class Browser—and that's what this chapter is about. Before we discuss these tools, however, we should cover what a class is, what OOP means to Visual FoxPro, and so on.

This chapter consists of three parts. The first will be a light introduction to OOP. It's not an exhaustive tutorial on all of the details; rather, you're going to learn enough to discuss classes and the class tools effectively. There are two reasons for this light treatment. First, this section in the book is about the tools, and we're going to cover just enough theory in order to learn how to use the tools. Second, OOP is complex and hard to understand. You have to learn the lingo. It takes more than one pass at it, and simply reading the same chapter twice isn't enough. So I'm going to repeat this information, in greater detail, ad nauseum, perhaps, in the Sample

---

Application section of this book. (It's a bit like learning a foreign language: You have to hear it over and over before you begin to think in the language.)

Just as development of an application is an iterative process, so is learning about object-oriented programming. You learn a little bit and get comfortable. Then you learn a little bit more, piling that on top of the stuff you just learned and have grown comfortable with. And so on and so forth. It's important to first learn the terminology and theory.

I remember sitting in the lounge late at night at a conference with about 20 other developers back in 1994 or so. One of the stalwarts was listening to several people go back and forth about the upcoming Visual FoxPro 3.0, and asked, "Do you really think we're going to start using 'polymorphism' and 'inheritance' in our daily conversations?" Another in the group replied, "Sure. Twenty years ago, I can remember the same question being asked about 'normalization' or 'tuples'—but those concepts are second nature to us now."

The second part of this chapter will deal with the tools themselves: the Class Designer and the Class Browser. Once you're comfortable with these concepts and techniques, you'll learn the tools you need to use in order to use object-oriented programming in Visual FoxPro.

Finally, I'll build a set of basic, but solid, base classes that you can begin to use and enhance in your own development, and show you how to build forms with them.

## A quick introduction to object-oriented programming

The first thing to understand about OOP is that there is no single way to do any of this. If you wander the halls of a conference on object-oriented programming, you'll hear all of the buzzwords and catch phrases—and then find different people using them to mean different things according to context, language, and platform. Once you get past that shock, you'll encounter a variety of people—from the pragmatists, those who have to implement real-world solutions today, to the theorists, whose mission in life is to evangelize about POOP—Proper Object-Oriented Programming. It's much like the Relational Database Management System purists who claim that today, in 1999, there still isn't a "true" relational database system commercially available. All along, however, we've been building normalized data structures, making do with the available tools, and running global businesses with these systems. While we may never reach the Holy Grail of OOP, we can give it our best shot and get 80% of the benefits with 20% of the worry.

Thus, as a result, the discussion that follows won't necessarily win any "POOP" awards or put me in the OOP Hall of Fame. This is my take on OOP as it relates to Visual FoxPro, at this stage in the lifecycle of VFP. I've been using and learning Visual FoxPro OOP over the past five years, and it's starting to sink in. I've even shipped an application or two. This chapter will give you the fundamentals. In 20 minutes, you'll understand what you need to know to take advantage of VFP's OOP capabilities. Once you're comfortable with the contents of this chapter, you may want to investigate Markus Egger's book, *Advanced Object Oriented Programming with Visual FoxPro 6.0* (Hentzenwerke Publishing).

There are a dozen or more fancy buzzwords attached to the OOP model, including class, object, property, method, inheritance, hierarchy, encapsulation, polymorphism, subclass, and so on. In my mind, the key concepts that will bring you immediate benefits are subclassing and inheritance.

## **The original form and control objects**

Visual FoxPro comes with a default form object that is used as a starting point every time you create a new form. It has a gray background, measures about 360 pixels wide by 240 pixels high, has a system-type border, is named “Form1” and has a caption of “Form1.” Additionally, VFP comes with about 20 native controls, from command buttons to hyperlinks. Each form and control has default properties and events as well. Each time you create a new form or place a control on a form, it seems like you are making a copy of the default form or control.

Actually, you are not making a copy—you are creating an object that consists of pointers to the original version of that form or control. The original is contained in the code of the VFP6.EXE file and can’t be changed. This object is called an *instance* of the form or control, and the process of creating the instance is called *instantiation*. You can, to an extent, think of the original version as the die (or mold) used to create multiple copies of a firing pin for a rifle, and each instance as an individual copy of that firing pin.

When you run your application, the form or control points back to the information in the original version in the .EXE. Because you’re always either running the application from within Visual FoxPro or from an executable that requires a Visual FoxPro component, the code that describes the form or control is always available.

## **Inheriting from the original**

Because the instances reference the original version, if you change the original version, each instance will reflect that change. (In fact, you see that behavior often as you use new releases of software. Ever notice that a new version of word processor has unintended effects on existing documents?) In fact, now that we’re using the third major release of Visual FoxPro, we can see this ourselves. In Visual FoxPro 3.0, the default form had a white background. In VFP 5 and 6, the default form had a gray background. If you ran a VFP 3 application in VFP 6, all of the white forms would automatically have gray backgrounds—without you lifting a finger.

This behavior is called *inheritance*. The instances of a form inherit properties, such as color, size, and captions, and methods such as Click, Drag, and Valid, from the original version. When you change the original version, as was done when you used VFP 6’s executable instead of VFP 3’s, the instances inherit those changes automatically. If inheritance existed in the rifle factory described earlier, when you changed the size or shape of the die, every firing pin created from that die would automatically morph to the new size or shape. Furthermore, you could think of a firing pin as having certain events—for example, when the pin strikes another piece of metal, it creates a spark. The firing pin’s Strike event initiates the Spark function. Again, if the firing-pin mold had inheritance, you could change the functionality of the die’s Strike event to Puncture, and all of the firing pins ever made from this die would then Puncture when they struck another piece of metal.

## **Creating your own originals**

But you’re not in a rifle factory. You’re sitting in a chair in front of a cathode ray tube with a stuffed Dilbert doll sitting on top of it, and you want to create an order-entry system. Here’s the essence of object-oriented programming in Visual FoxPro: You have the capability to create your own versions of these default forms and controls, so that subsequent forms you create (or controls you drop on forms) inherit properties and methods from the original versions you created. Then you can make changes to your original versions, and all of the forms and controls

---

you've created from those original versions will reflect those changes. You're no longer restricted to accept the defaults that came with the package. (Before I continue, I should mention that it's important to remember that the original versions you create still use Visual FoxPro's default forms and controls as *their* original definitions!)

How about a real-world example or two? When was the last time you created a form that had no objects on it? Pretty long ago, eh? Every form I've ever created in my life has had two buttons on it: a Help button and a Done button. And those two buttons always do the same things: The first brings up a help window where the topic displayed relates to the current form, and the second one cleans up everything and closes the form. Wouldn't it be nice if the next time, and every time after that, when you issued the command:

```
create form <name>
```

the form that appeared already had those two buttons on it, and those two buttons had the behavior you defined? You can do this simply by creating your own default form, dropping two buttons on it, and then telling VFP to use that form as the default instead of Visual FoxPro's default. Furthermore, if you realized that you needed to change the behavior of one of those buttons, you could do that once, in your default form, and every form in the application would then inherit that behavior automatically.

For example, suppose you decided that you wanted the Help button to work one of three ways—for most forms, it would bring up a context-sensitive Help window; for a small percentage of forms, it would bring up a user-defined Help screen; and for the last few, it wouldn't be visible at all. You could create a property of the form that indicated which type of behavior the button should have, and then control the behavior of the button in the default form. From then on, every form that was based on your default form would have those three behaviors available, depending on how the property was set.

You can also create your own default controls. For example, in Chapter 9 I discussed the List Box and Combo Box controls, and indicated that I thought the best way to populate them was to include a custom property of the control that was an array, and then to set the ControlSource of the list box or combo box to the contents of that array. Having to add an array property to every list box and combo box would be a nuisance—if indeed you could even do it. If you define your own List Box and Combo Box controls, however, you can add your own properties and methods, and they'll be available each time you drop instances of those controls on a form.

## OOP terminology

Now it's time to introduce the real terminology for these pieces and processes. The definition of one of these objects, be it a form or a control, is called a *class*. When I created a form with a Help button and a Done button on it that I was going to use as the default to create more forms, I created a *form class*. When I created a list box (with some of my own custom properties and methods), which I was going to use as the default when I needed a list box to put on a form, I created a *list box class*.

The default forms and controls that come with Visual FoxPro are definitions for each of those objects, and they are the *Visual FoxPro base classes*. After you create your own forms and controls to use as the defaults instead of Visual FoxPro's base classes, you'd refer to your

creations as *your base classes*. The act of creating a class that references another class is called *subclassing*, so when you created a list box with an array property, you were subclassing Visual FoxPro's List Box control.

When you create a form from a class, you are creating an *instance* of that class—or, as mentioned earlier, *instantiating* the class. The form is the instance, and the class from which the form was instantiated is simply called the form's *parent class*. If you subclass a class, the class higher in the hierarchy is also called the parent class. Some languages, by the way, call the class higher in the hierarchy the *superclass*.

I've just described creating your own form base class with a Help button and a Done button. What if you need a form that isn't supposed to have a Help button or a Done button? You might be thinking that the best solution would be to create a complete set of your own base classes that are no different from Visual FoxPro's base classes, and then create a second set of classes that were based on your base classes. That way, you'd have a form base class that was completely generic, and you could subclass your form base class, creating a form class with a Done button and a Help button. If, at some point, you needed a form that didn't have a Done button or a Help button, you could simply subclass your form base class, and create a second form class that had the attributes or behaviors you desired.

Thus, you'd have a hierarchy that looked like this:

```
VFP form base class - Your form base class - Your Help/Done form subclass  
- Your Other form subclass
```

with two subclasses both inheriting from your form base class. The philosophy here is to never directly create instances from your base class; instead, you create instances from subclasses of your base class. Your base class, then, is referred to as an *abstract class*—one that is never directly used. This is a valid approach, and you might feel most comfortable following it.

You can create subclasses of subclasses as many times as you like. This hierarchy of classes is called a *class hierarchy*, which looks much like a set of routines that call subroutines that in turn call other subroutines. Of course, before you get too carried away, dreaming of class hierarchies dozens or hundreds of levels deep, remember that in the real world you have to maintain those classes, and you also have to make your applications perform on computers with limited resources. Digging through a class hierarchy 10 or 15 levels deep will require more horsepower than a class hierarchy only one or two levels deep, and will end up being very difficult to use, as you try to determine the differences in behavior from one level to another.

## **Inheritance and overriding methods**

I've discussed the idea of inheritance several times, but there's one more item to mention before moving on. As you've seen, you can create a class and put code in its methods. Then, once you create an instance of that class, the code that is in the methods of its parent is also available to the instance. Suppose, in the form with the Done button, there was code in the Release method of the form to perform certain housekeeping chores before the form is actually completely closed. You might want to check for data that has been entered in the form but not yet saved, for example.

When you instantiate a form based on that form class, you'll see the Done button, but if you look in the Release method of the form instance, there won't be any visible code. This is to be expected if you think about it for a minute. When you create a form based on Visual

---

FoxPro's form base class and you open up a method, you don't see any code in it, do you? However, certainly Visual FoxPro code gets executed, right? For example, if you look in the GotFocus event, you don't see code, but when GotFocus is fired, things happen, such as the color of the title bar changing. The code is stored in the base class—in this case, Visual FoxPro's VFP6.EXE.

This is good, although I've heard this described by idiots as a downside to OOP. It means that you're not carrying around a lot of excess baggage, but even more so, it means that because the code being executed is stored only in the base class, changes need to be made only to the base class. If that code also ended up in the instance, changes made to the base class would either have to be manually propagated to every instance, or they wouldn't be reflected in the instances. In other words, no inheritance. I've heard rumors that Visual Basic operates like this, but I can't believe anyone would be so foolish to actually use it, given such a lame feature.

There might be times, however, when you don't want the inherited behavior from the parent class. For instance, in the example of the class with code in the Release method, you might need different behavior than what was provided in the class definition. Or you might just not want the code executed at all. In both cases, this is referred to as *overriding* the method code in the class definition.

## Non-visual classes

Now that you're a little comfortable with the concept of classes, I'm going to throw you a curve. As you've seen, a class can be a form, from which you can create multiple instances that all inherit the properties and methods of the class. It can also be a control, and you can drop instances of the class on a form as needed. Finally, a class can also be a group of controls that act together as a unit; just like the class definition of a single control, you could drop an instance of the control class on a form as needed.

So it seems that classes are “things” that you can see, move around, and otherwise manipulate. The next conceptual leap to make is that of non-visual classes, and I've found these are harder to grasp for many people. A class can have a non-visual nature—in other words, there is no object to call from a menu or to move around on a form. The question that immediately comes to mind is “So what is a non-visual class, and, more importantly, what good is it?”

When you began creating applications in the very olden days, there was no such animal as a “form” or a “report”—instead, you used programming commands to draw discrete elements on a computer display console. These programming commands resided in files of two types. The first type was a custom-written program, while the second was a library. The library included programs that could be reused throughout an application, and, indeed, across applications.

In the early 1990s, the idea of “forms” (some development environments called them “screens”) and “reports” was born, and then a programmer had two types of elements to work with—programs that still consisted of commands, and forms and reports, which were visual representations of programs. Programmers still used libraries of commonly used programs.

With object orientation, these libraries can be moved into non-visual classes. An entire application, and indeed, multiple applications, can still access these libraries, but by placing them into classes, another advantage accrues: The library can be subclassed, just like a form or control.

This still hasn't quite answered the question, "So what is it?" Remember that a class has properties and methods. Properties, again, are really just variables that are initialized with respect to the class. A form that allows you to search for the name of a company might have a NameToSearchFor property. When the form is started, that property might be an empty string, but it would eventually be populated with the value that a user enters into a text box. Methods, as well, are just functions that belong to the class. When the class is a form or a control, those methods are easy to visualize—Init, Click, Destroy, and so on.

You can create a class that has properties and methods but doesn't have a form or a control to go along with them. What could this be? Think of a function library that has a number of procedures that do date and time calculations. You would have variables that are initialized, a number of subroutines that perform operations and return values, and—hey, these sound a lot like properties and methods, eh? One variable might be the initial date sent to a function; another might be a value that deals with the type of format required. And one function in this library might simply convert any type of input to a common format, while another function might return error messages if a parameter passed to the function was invalid.

A non-visual class could do the same types of things—using properties to store parameters sent to a method in the class, and then various methods of the class to validate incoming parameters, throw error messages, and so on.

Another example of a non-visual class would be one that handles the environment of the machine during application startup. One of the properties might be the default directory at startup, while a method in this non-visual environment class would check for available memory or disk space before loading the rest of the application.

A non-visual class can be subclassed just like any other class, and its methods can be overridden in an instance of the class, just like visual classes.

So a non-visual class can be thought of as a function library. But we can take this considerably further. As you get deeper into the world of Visual FoxPro applications, you'll start to see the term *application class*, as in "The XXX application class provides the framework for all of our custom applications." Your mind starts reeling again. What in the world is an application class?

Many developers have created a core set of programs that provide a generic foundation upon which they can duplicate and add components as required. This foundation provides a series of services, such as file handling, user security and permission levels, error trapping, and so on. All of these generic services should "come along for the ride" for every new application. You can think of this as a skeleton or framework upon which a custom application can be built. This skeleton consists of one or more programs, some of which might be located in a common library.

In the Visual FoxPro world, an application class is often a non-visual class that contains all of the properties and methods (remember, these are just variables and subroutines) that make up this framework. Depending on the complexity of applications and the requirements of the developer, an application class might actually contain both visual components (such as a logon screen or a user-maintenance screen) and non-visual components (such as a routine to handle missing files or to do an environment check).

## Where OOP fits in with Visual FoxPro

I hope you're getting a little anxious by now. I know that after I'd heard all the explanations without having gotten my hands on the product yet, I was bursting with questions like:

- “Where is a class stored?”
- “What does a class look like on disk?”
- “How does inheritance actually work in an application?”
- “I’m in Toledo, but I’ve got an application running on a machine in Madrid. How does my application in Madrid know that I’ve changed a parent class in Toledo?”

Most of the books that discuss OOP use examples such as blueprints, telephones, light switches, and red circles. As far as theory goes, great. But the first thing I asked was, “A class has to be stored in some sort of file, right? What does that file look like? Is it a text file? Binary gobbledegook? A table? Some new sort of structure? And where is it stored on disk?” And then I wanted to know exactly how VFP made inheritance work. Just as changing the original mold from which firing pins were being manufactured wasn’t automatically going to change all of the firing pins that had already been made, I wasn’t quite sure how the changes I made to this class file on my machine were automatically going to be reflected in the various applications that relied on this class file. It seemed like an awful lot of magic.

The answer is really quite straightforward, and reading back over my words of five years ago, it now seems pretty obvious—but it sure wasn’t then. And I’m sure the implementation in C code had one or two programmers at Microsoft working past 5 p.m. once in awhile.

First, as you know, the representation of a form on disk is a table. The form itself is represented by a couple of records, and then each control on the form is represented by an additional record in the .SCX file. Indeed, if you haven’t before, you can open a form’s file as a table just like any other .DBF file—you just need to include the extension. The following code will show you the inside of a form:

```
use MYFORM.SCX
browse normal
```

Be careful, though—Visual FoxPro expects certain items in certain places, and if you change or delete things accidentally, you might not be able to open the form in the Form Designer again.

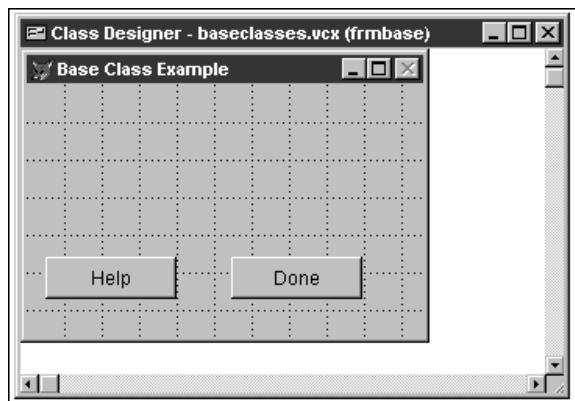
A class is just another table—with the exact same structure as an .SCX table, but with a .VCX extension. Just as a regular form has records that map to the controls on it, and fields in each record that describe the properties and methods of the form or a control on the form, a form class has records that map to the controls on the form, and fields that describe the properties and methods. In addition, other fields in the .SCX table point to .VCX files, and to specific records in the .VCX file. Kinda like a parent-child relationship, eh? And when you change the class—in other words, change the records in the .VCX—the next time the form that references that class is compiled, the changes in the .VCX will be taken into account. Rather simple, isn’t it? If you’re curious (and careful), you can open a .VCX file just as you can open a form’s .SCX file.

Thus, when you ship an application, you need to include the classes—the .VCX files—along with your menus, forms, reports, and other files. Or, if you’re building an executable file, the build process will compile the classes used in your application just as it does ordinary programs.

Thus, here’s the answer to the question, “How do changes to a class library get reflected in the application?” When you change a class, you must ship the new .VCX or a new .EXE that reflects the new .VCX to the location where the application is running, just as when you updated a procedure library file.

I’d like to discuss the contents of the .VCX and .SCX files in a bit more depth before moving on, using a real example to illustrate where all the pointers go.

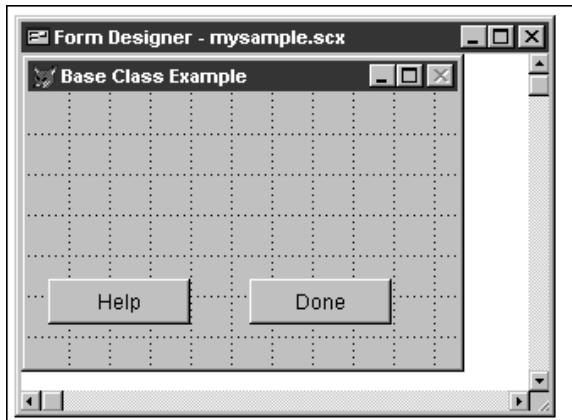
Suppose your form base class consists of a Visual FoxPro base form and two command buttons, both from a Visual FoxPro command-button base class, as shown in **Figure 10.1**. The form base class is named frmBase, and the VCX that holds this form class is named BASECLASSES.VCX. If you open BASECLASSES.VCX, you’ll see five records. The first is just a placeholder. The next three records represent the three objects that make up this class: a VFP form and two VFP command buttons. If you examined the Class field of these three records, you’d see they reference Visual FoxPro’s built-in objects. The last record is another placeholder that you shouldn’t worry about. The key point is that this class definition consists of three records.



**Figure 10.1.** A sample form base class consisting of a Visual FoxPro form and two Visual FoxPro command buttons.

Next, suppose you were to instantiate a form from this class (don’t worry about exactly how that process works quite yet), and name this form MYSAMPLE.SCX, as shown in **Figure 10.2**. If you opened MYSAMPLE.SCX as a table with the USE command, what would you see? If you had created a regular form, you’d expect to see three records—one for the form and one for each of the command buttons, right? But we created this form from a class, which means that this form is essentially just a pointer to a class. That means you’re going to see only one record in the .SCX file. The Class field in that record is going to point to frmBase, and the ClassLoc (class location) field in that record is going to point to BASECLASSES.VCX.

(Actually, the .SCX contains several other records—placeholders similar to those found in the .VCX file as well as another record for a DataEnvironment. But those records aren't germane to our discussion.)



**Figure 10.2.** A sample form instantiated from frmBase.

Finally, you might be wondering about the relationship between frmBase and BASECLASSES.VCX. A .VCX file is called a class library, and much like a regular library can contain more than one book, a class library can contain more than one class. Thus, frmBase is simply one class that is located in BASECLASSES.VCX. You could create a list-box class, lstBase, and store that in BASECLASSES.VCX as well. Then the .VCX would have four records (in addition to the placeholders I mentioned earlier). The first three would make up the form base class and the fourth would be the list-box class. Because a list-box class contains only one control, it needs only one record.

This class library, BASECLASSES.VCX, and the associated form, MYSAMPLE.SCX, are both included in the source code downloads for this book. You can open them and spelunk through the data to your heart's content.

## Quick start to creating classes

You're probably plenty full of theory by now. It's time to actually start creating your own classes. In this section, I'll cover creating classes both from forms and from controls, and then show you how to use both of them together.

### Form classes

#### Creating your own form base class

1. Make sure that an active form base class hasn't already been subclassed from Visual FoxPro's base classes.
2. Look in the Forms tab of the Tools, Options dialog. The Form Set and Form Template Classes text boxes should be empty and the check boxes should be unchecked.
3. Create a form by issuing this command:

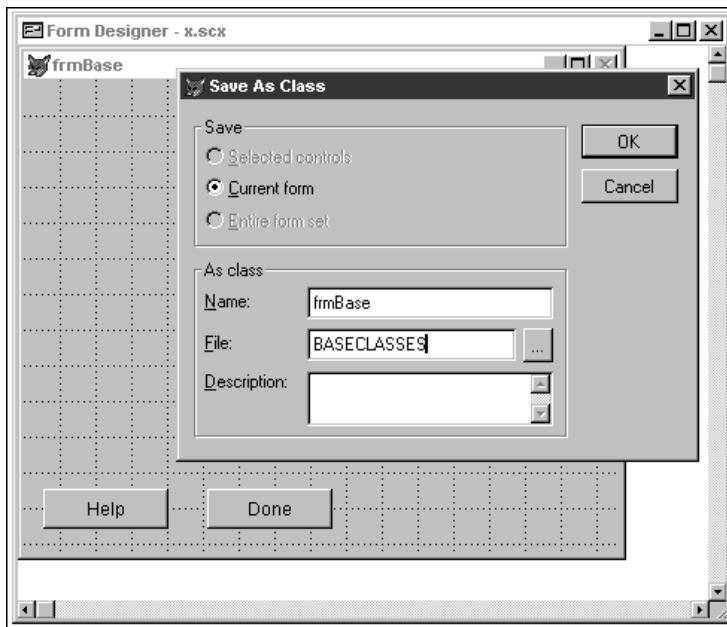
```
create form X
```

4. Modify the form to reflect how you want all of your forms to look and behave. Here are some ideas:
  - Change the caption to frmBase. This will act as a visual clue that the form you just created has come from your own base class—not from Visual FoxPro. And more importantly, the reverse. It's pretty discouraging to create a form, drop a bunch of controls on it and work with it for a while, only to discover that you're working on a Visual FoxPro base class form and not your own base class form.
  - Drop a couple of Visual FoxPro buttons on the form – such as for Help and Done (or Close, if you prefer.) Note that when you're doing this for real, you'll most likely want to use controls that you've subclassed yourself, not Visual FoxPro's controls.
5. Select the File, Save As Class menu option. The Save As Class dialog opens as shown in **Figure 10.3**. (Note that if you select Save As, you'll just create another form that references Visual FoxPro's base class.)
6. Enter the name of the class—frmBase—in the Name text box, and the name of the class library—BASECLASSES—in the File text box.

Note that Visual FoxPro will append the .VCX extension to BASECLASSES automatically, because, after all, it knows you are creating a class.

You can also choose an existing class library by clicking the ellipsis button to the right of the File text box, and then selecting a .VCX from the list that appears or navigating to the .VCX you want.

7. At this point you've created a .VCX file, but you've also still got an .SCX file on the screen. You can get rid of it by closing it and not saving the results.



**Figure 10.3.** The Save As Class dialog allows you to convert a form into a class definition.

### Using your own form base class

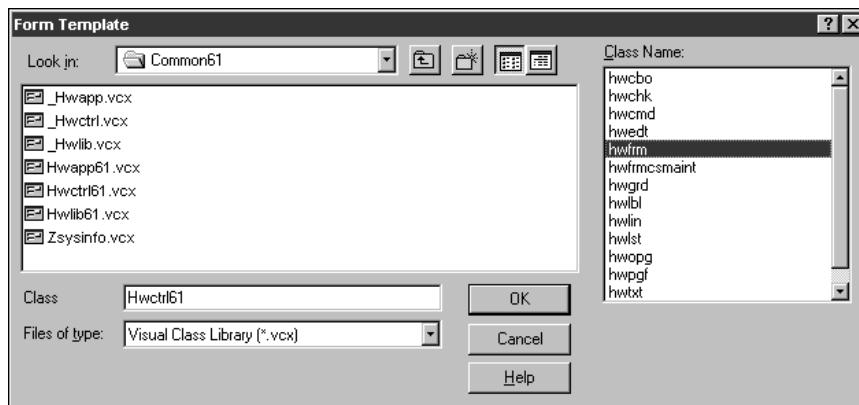
So now you've got your own form base class. How do you use it as the parent class for all forms you build from now on? If you've tried to create another form, watching out for some "create form from user-defined base class" option that you thought might have previously escaped your notice, you'll find nothing new.

To use your new form base class, follow these steps:

1. Open the Forms tab of the Tools, Options dialog. The Form Set and Form Template Classes text boxes should be empty and the check boxes should be unchecked, since you just verified this when you created your form class in the previous section.
2. Click the Form check box in the Template Classes control group. The Form Template dialog will open, prompting you to select a form class.

This can be a bit tricky the first few times you use it. Because it looks much like a regular File, Open dialog, you might be tempted to select a .VCX file and then immediately click OK. Resist that temptation. Note that the Class Name list box on the right side of the dialog becomes filled with the classes in the .VCX that you chose, as

shown in **Figure 10.4**.



**Figure 10.4.** Be sure to select the appropriate class in the *Class Name* list box when picking a Form Template class.

If you just click OK, you'll end up picking the first class in the .VCX—most likely not what you wanted.

Many developers keep all form and control classes in the same .VCX. Yes, the .VCX can get rather large, but because it's likely that you're going to use multiple controls on a form, you might as well keep all of them in the same file so you have to open only one class library.

3. Once you've selected the form class you want, click OK and you'll be returned to the Tools, Options dialog.
4. Click OK if you want this form class to be used as your default form-class template for the current Visual FoxPro session, or click Set As Default if you want this selection to persist past this session.

### **Creating a form from a Form Class template**

Once you've set up a Form Class template, creating forms based on that class is no big deal.

1. Create a new form by issuing this command:  
`create form Y`
2. You'll see that your new form has the characteristics of the class you used as your Form Template class.
3. Open the Properties window and examine the Class and ClassLibrary properties. You'll see that they have read-only values of the class/class library you specified.

## Using a different form base class “on the fly”

As you become more experienced, you’ll find that you don’t want to use a single form class as the parent for all of your forms. You might have one form class for minor entity maintenance, another for dialogs, a third for query screens, and a fourth for a specific type of data-entry form. It would be pretty darn inconvenient if you had to go through the rigmarole of having to change a Tools, Options setting each time you wanted to create a different type of form. Fortunately, you don’t have to.

The command CREATE FORM now has a clause that allows you to specify a class in a specific class library, like so:

```
create form Z as hwFrmQuery from BASECLASSES.VCX
```

This command will create a form named Z and use the hwFrmQuery class in the BASECLASSES class library. Pretty slick, eh?

## Creating form subclasses

In the last section, I suggested that you would likely end up with multiple form classes in your class library. However, for the sake of expediency, I took a shortcut when providing examples: each of the classes I named was created from the Visual FoxPro form base class. While technically you could do that, it would be a bad idea. Bad! Bad! Bad!

Rather, you would want to create a form class on which all of your other form classes would be based. You might not even use that first form class to create actual forms, but only to serve as the parent class of the form classes that you *will* use to create actual forms.

This might feel a little vague, so an example may help. My form base class is named hwFrm, but I never create a form from it, like so:

```
create form ZZZ as hwFrm from BASECLASSES.VCX
```

Instead, I create more form classes using hwFrm as the class definition, not Visual FoxPro’s form base class. These classes would have names like hwFrmDialog, hwFrmQuery, hwFrmMaint, hwFrmMaintME, hwFrmMaintCommon, and so on. If you look carefully, you’ll see that you can infer a three-level hierarchy here:

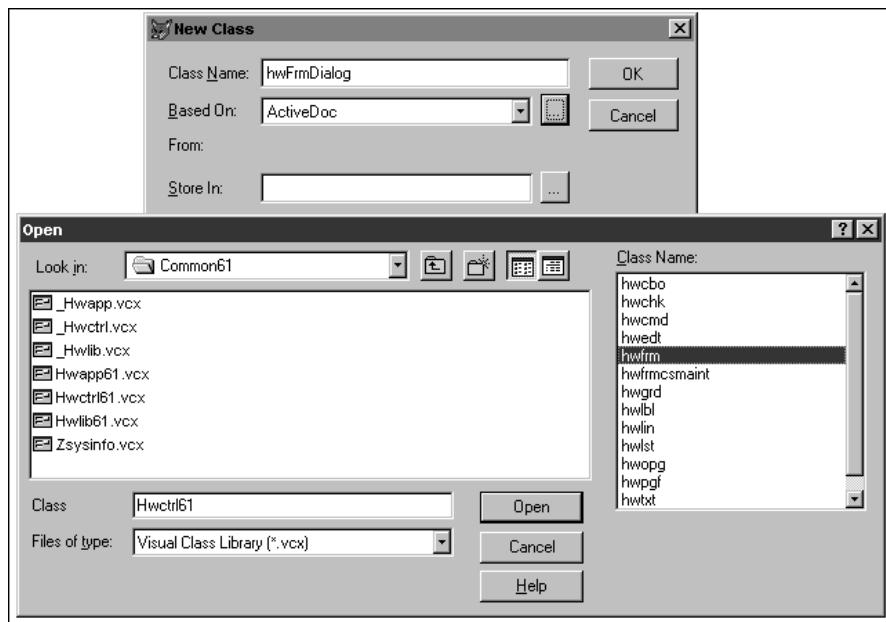
```
hwFrm - hwFrmDialog  
      - hwFrmQuery  
      - hwFrmMaint - hwFrmMaintME  
                    - hwFrmMaintCommon
```

The three form subclasses—hwFrmDialog, hwFrmQuery, and hwFrmMaint—all have hwFrm as their parent. Then, hwFrmMaintME and hwFrmMaintCommon both have hwFrmMaint as their parent. The subclasses hwFrm and hwFrmMaint are not used for creating forms directly; they’re used simply as definitions for further classes. If you remember the definitions above, these are both abstract classes.

So how do you go about creating hwFrmDialog from hwFrm? Follow these steps:

1. Identify the class upon which you want to base a subclass, such as hwFrm in BASECLASSES.VCX.

2. Issue the CREATE CLASS command (or select the File, New, Class menu option). The New Class dialog appears, as shown in the background in **Figure 10.5**.
3. Enter the name of the new class, such as hwFrmDialog, as shown in the New Class dialog in Figure 10.5.
4. Click the ellipsis button next to the Based On combo box.
5. The Open dialog will appear, as shown in Figure 10.5.



**Figure 10.5.** Selecting a class upon which to base another class.

6. Select the class upon which you want to base your new class. In Figure 10.5, hwFrmDialog will be based on the hwFrm class in the HWCTRL61.VCX class library.
7. Click Open.
8. Choose (or create) the class library in which the new class will be stored. I suggest that you store your new class in the same class library that holds the parent class.

Note that you *could* choose a different class library to store hwFrmDialog. You would want to have a specific reason to do so. The downside of using a second class library to store hwFrmDialog is that you'd then have to open two class libraries in order to use a single class—why make it that hard on yourself? On the other hand, what if you wanted to distribute hwFrmDialog to someone else, without sending them every class you ever created (that also happened to be in that class library)? This is one of the toughest design decisions to make: how granular to make your class libraries.

## **Registering a form class with Visual FoxPro**

As you know, you can select controls from Visual FoxPro's base class by grabbing them from the Form Controls toolbar. You can create your own toolbar that contains buttons for your own classes and grab those classes to create forms and place controls on forms. There are two ways to do this.

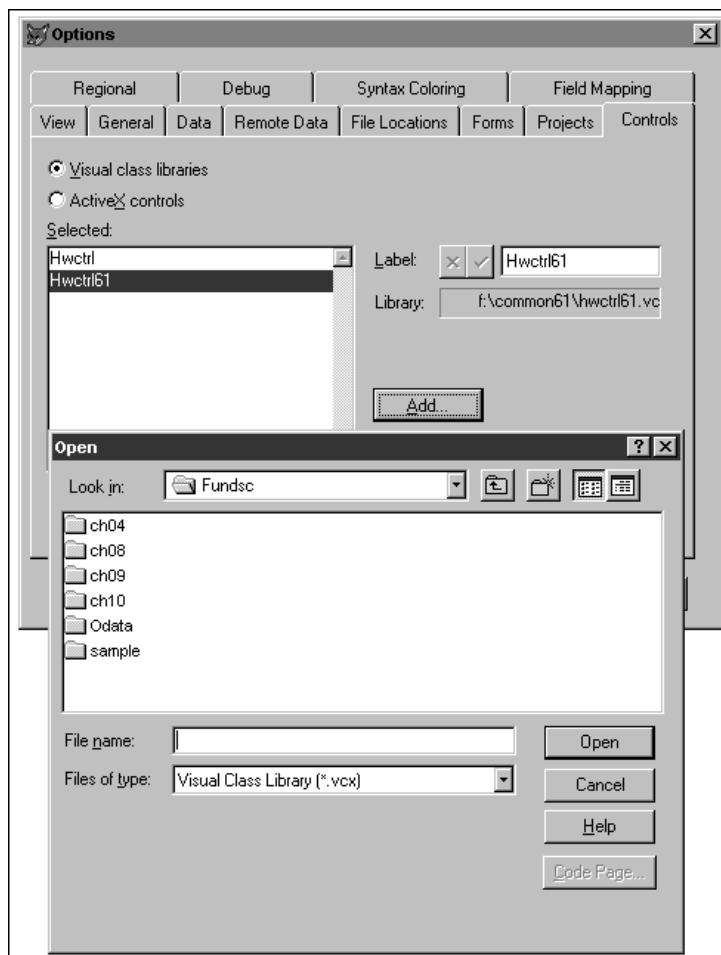
### ***Use the Add button from the View Classes menu***

1. The View Classes button on the standard Form Controls toolbar looks like three books and has a very small arrow in the lower right corner. Click it.
2. A context menu with three commands appears. Select the first command: Add.
3. A standard Open dialog opens, prompting you to select a class library file. Select the class library of your choice.
4. The contents of the Form Controls toolbar will now change. The first two buttons, Select Objects and View Classes, and the last two buttons, Builder Lock and Button Lock, will stay the same. In between, however, the Visual FoxPro base class control buttons are replaced by buttons representing the classes in the class library you chose.
5. When you move your mouse over a toolbar button, a tool tip displays the name you gave that class.
6. When you click the View Classes button again, you'll see that your class library name has been added to the context menu.

You can continue adding class libraries to this context menu by following the previous steps, and thus be able to switch between class libraries quickly and easily.

### ***Use the Controls tab of the Tools, Options dialog***

1. Open the Tools menu and select Options, then Controls. See **Figure 10.6**.
2. Select the Visual class libraries option button.
3. Click the Add button.
4. Select the name of the class library (remember, it's a .VCX).
5. Click OK to register this library for this session, or select Set as Default to register this class library for future sessions.



**Figure 10.6.** You can register a class library using the Controls tab of the Tools, Options dialog.

### Attaching a custom icon to a registered class

If you have several forms in the same class library, as you ought to, you'll see that your toolbar now has four or five (or a couple dozen!!!) buttons—each with the exact same icon. You don't have to settle for this boring panoply of computerdom, however. You can attach a different toolbar icon of your own choosing to every class in the library. Here's how:

1. Bring up the Class Designer with this class by issuing the MODIFY CLASS command or opening the File menu and selecting Open, Class menu. The Class Designer appears.

- 
2. Select the Class, Class Info menu option.
  3. Select an icon for this class by clicking the ellipsis button next to the Toolbar Icon label. You can draw your own, or if you'd rather leverage the work of others, you can find a slew of sample icons in the <location of VFP>\MISC\BMPS directory.

### Modifying a form class

Once you've created a form class, you might find you want to modify it. You can simply use the MODIFY CLASS command, like so:

```
modify class hwFrm of <path>\MYBASECLASSES.VCX
```

I find, however, that as I get older, I want to type less and less. If you're inclined along the same lines, you can open a class by using MODIFY CLASS without any further parameters. Doing so will bring up the Open dialog pictured in Figure 10.5. Locate the .VCX that contains the class you want to modify, and then be sure to select the class itself in the Class Name list box on the right.

## Control classes

### Creating your own control base class

Now that you've got the hang of creating your own form classes, you'll want to create your own control classes. The procedure is similar to that of creating form classes:

1. Issue the command CREATE CLASS or select the File, New, Class menu option. The New Class dialog appears.
2. Type the name of the class you are creating.

Again, remember that naming conventions will make your life easier down the road. Consider using the first three characters of the naming convention for controls on forms, such as cbo for Combo Box and cmd for Command Button. I personally use names like hwCmd, hwChk, and so on (where "hw" stands for Hentzenwerke).

3. Select the object you are using as the parent class for your new class.
4. Select the name of the class library into which this class will be placed, or enter a new class name by typing in the File text box.

Remember, classes are contained in class libraries, and class libraries are files with .VCX extensions. The name you select depends on what else you're going to put in the library. For now, use a name like BASECLASSES.VCX.

5. Click OK. The Class Designer appears with a copy of the control in the window. Because you are creating only the control, resizing the window will change the default size of the control, so be careful.
6. Make modifications to the control as desired.

For example, you might want to change the background color of certain controls such as check boxes and option button groups so that they match the background color of the form base class you created in the previous section. If you don't, the background of the control will show up against the form—the programming version of wearing a colored t-shirt or bra underneath a white dress shirt or blouse.

You might also want to change the default caption of the object. I change the caption of all of my base-class controls to conform to the Visual FoxPro conventions so that when I drop the control on a form, it's obvious that the control is one of mine instead of a VFP base class. For example, the default caption for my label class is "lbl", and for my check-box class it's "chk."

7. Save the class.
8. Repeat the previous steps for every control you want to add to your base class. Be sure to specify the same class library name.

### **Creating control subclasses from your base class**

Just as you created subclasses from your form base class, you will most likely want to create subclasses of your control base class. This mechanism works the same way as subclassing your base form. For example, you might want two types of text boxes—an editable text box and a read-only text box. I would name mine like so:

```
hwTxtEditable  
hwTxtReadOnly
```

Actually, I'd abbreviate the names because when I drop a control class on a form, I name the instance by appending a specific identifier to the class name. Thus, a pair of text boxes for a person's name would look like this:

```
hwTxtEdNameFirst  
hwTxtEdNameLast
```

If they were read-only text boxes, they'd look like this:

```
hwTxtRONameFirst  
hwTxtRONameLast
```

### **Registering your controls base class**

You can register your controls base class the same way you registered your form base class. Remember to use the Set as Default button in the Controls dialog (Tools, Options, Controls) in order to keep the registrations available from session to session.

## Placing controls on a form from your base class or a subclass

Now that you've got your controls base class, and probably several subclasses of your form base class as well, you're going to want to create controls on forms from those classes. This is done a little differently than with forms—and fortunately it's easier, because you'll be creating more controls than forms. As with all things FoxPro, there is more than one way to do this.

### Use the Project Manager

1. Add the control class to the Project Manager:

- Open the Project Manager.
- Select the Classes tab or the Class Libraries icon in the All tab.
- Click the Add button.
- Select the class library that contains the class or subclass from which you want to create controls. The class library is added to the Project Manager.

Note: You can select the outline control (the plus sign to the left of the library icon) to expand the class library and see the classes contained in it.

2. Create a blank form or modify an existing form.
3. Drag the class (not the class library) from the Project Manager to the form. The control will be placed on the form.

### Select the class from the Form Control toolbar (after registering the class)

You must register the class from which you want to create controls. (See the section "Registering a control class with Visual FoxPro.") Then follow these steps:

1. Create a blank form or modify an existing form.
2. Select the View Classes icon on the Form Control toolbar and select the menu option for the class library that contains the class from which you want to create controls. The buttons for that class appear on the toolbar.
3. Click the button for the control class from which you want to create a control. The button appears depressed.
4. Move the mouse and click on the Form Designer. An instance of the control class will be placed on the form.

## Control and container classes

Now that you have your control base class, and probably several subclasses of your base class as well, you might want to create a class that consists of several controls but without using a form to "hold them together."

How to make this happen? You can do this one of two ways.

The first is to create a container class. A container is an object that contains other objects, and those other objects can be addressed at runtime. For example, a page frame and a grid are

both containers. A grid contains one or more columns, and you can address both the grid as well as each individual column in the grid. (And because a column is also a container, you can address the objects in the column—the header and the text box, for example.)

An example of a container would be a custom control that replaced a combo box. You could put a text box and a command button in a container, and allow the user to either enter a value in the text box or click the command button to “open up” a pick list of one sort or another.

The second way is to create a Control class. Note that I am capitalizing the word “Control” here so you don’t mistake this with the term “control class” that I have been using for the last umpteen pages.

The objects in a Control class can only be accessed individually while in the Class Designer—they are not available when a form is being designed or at runtime. You can think of a spinner or a list box as being a control with multiple components. A spinner has a text-box component, an up-arrow component, and a down-arrow component. You can access only the entire control—you can’t change the color of the up arrow to be different from the color of the text box, nor can you access a method of the up arrow; you access a method of the Spinner control.

An example of a Control class that you would make yourself would be a class that you want to distribute to others but that you don’t want modified. Just as you can’t get to the internals of the native Visual FoxPro spinner control—you can access only the interface provided to you—you might not want the user of your class to access anything but the interface you’ve provided.

The steps to create a Control or container class are essentially the same as creating a form: First you create the container and then you add controls to it. Remember that the controls you add to a Control class can only be modified here—and can’t be changed once you’ve instantiated the control on a form!

1. Issue the command CREATE CLASS or open the File menu and select New, Class. The New Class dialog appears.
2. Type the name of the class you are creating. For fear of being so redundant that you tune me out, remember that naming conventions will make your life easier down the road. The naming conventions for Control and container classes use the first three characters of *ctl* and *cnt*.
3. Select either a Control or container class in the Based on combo box.
4. Select the name of the class library into which this class will be placed, or enter a new class name by typing in the File text box.
5. The Class Designer appears with an empty box in the Designer window. This is the holder (I hesitate to use the word “container” for fear of confusion) for all the objects you will place in the class.
6. Add the controls and modify them as desired.
7. Save the class.

## Working with Visual FoxPro's object orientation

Learning how to use the tools can take you only so far. Once you start actually building your own classes and start using those classes in real-world applications, you're going to run into a variety of issues that aren't directly addressed in the directions for the tool. Here are some situations you're likely to run into.

### Naming classes

There are several schools of thought with regard to naming classes.

My preference is to name classes from the generic to the specific, as you saw with the sample forms I described in an earlier section. The first few characters act as a common identifier so that my classes don't conflict with someone else's—if three developers named their classes “BaseForm” and kept them in a library named “BASECLASSES.VCX”, it would be rather difficult to use them in concert.

The second batch of characters identifies the type of object, based on Microsoft's standard naming conventions, where “frm” stands for “form,” “lbl” stands for “label,” and so on. The type of class comes next. For example, for forms, you might have Dialog, Help, Maintenance, Data Entry, Query, and other types of forms. You might even break these down further, with multiple types of Maintenance or Query forms. There are no hard and fast rules about making decisions regarding these types of breakdowns, but you probably want to think along the lines of behavior and functionality, rather than appearance, when doing so.

The other popular naming convention goes the other way. I first encountered this convention when working with David Frankenbach on a Visual FoxPro 3.0 book on object orientation. His preference, coming from experience with other object-oriented languages like Visual C++ and used by the Microsoft Foundation Classes for Visual C++, was to prefix the specialization to the general class, instead of adding the specialization to the end.

Thus, he would name a hierarchy of text box subclasses like so:

```
TextBox (Visual FoxPro base class)
cTextBox (His custom base class)
ReadOnlyTextBox (A subclass of his base class that is read only)
```

Furthermore, David uses the three-character abbreviations to indicate that a class is functional, not abstract. Thus:

```
ReadOnlyTextBox
```

would be an abstract class while:

```
CmdReadOnlyTextBox
```

would be a functional class that you could drop on a form.

I'm not arguing that one is better than the other; but because I'm comfortable with my style, I'll use it throughout this book. If you prefer the other style, please free to use it. Or create your own convention if you like. Just be sure to use it regularly and consistently.

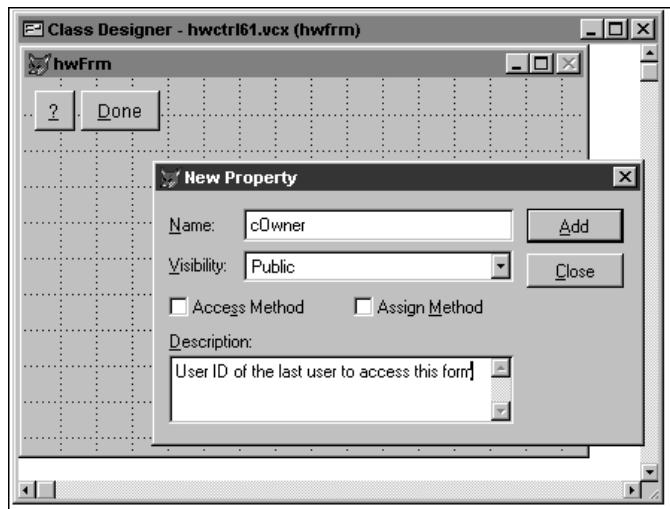
## Adding your own properties and methods

One capability of Visual FoxPro's toolset that escapes most people is the ability to add properties and methods to a class or a form. Remember that a property is simply a variable that is carried along with the object (the class or the form), and it isn't visible when that object hasn't been instantiated. You're familiar with the idea of declaring variables "local" or "private" within a subroutine—this does the same thing, only better.

Furthermore, a method is a segment of code that can be accessed only when the object is instantiated. The analogy to procedural programming here is not having access to a function in a function library until you've SET PROC TO.

This tying of properties and methods to an object is called *encapsulation* and, when done properly, forms the basis for object-oriented programming. Instead of having an application made up of module after module of spaghetti code, you'd simply have an object send a message to another object, referencing that second object's properties and methods. For example, when the Customer object (perhaps a form) needed to know the total sales for a customer, it would send a message to the Sales object, requesting that the Sales object run the TotalSales method, using the ThisCustomer property to do so. Yes, if you're used to procedural programming, this might sound a little freaky, but once you get the hang of it, you'll likely agree that it's much cleaner and less error-prone than the Old Way.

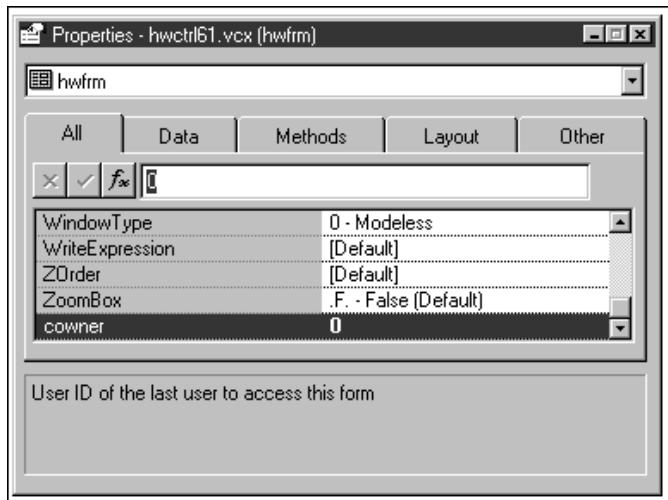
To take advantage of these properties and methods, we need to be able to add them ourselves. The process is extremely simple: Just select the Class, New Property or Class, New Method menu option (there are similar menu options under the Form menu when you're working with the Form Designer), and enter the name of the property or class as shown in Figure 10.7.



**Figure 10.7.** Adding a new property to a form class.

Once you've added a property or method, it will appear at the bottom of the Properties window. You can initialize a property to a value or data type in the Properties window as

shown in **Figure 10.8**, and you can add code to the method via the Code window. Notice that the description for the property or method appears in the bottom of the Properties window. You did enter a description, didn't you?



**Figure 10.8.** You can initialize a custom-added property in the Properties window.

You can edit or remove a property or method from the class by using the Edit Property/Method menu option under the Class (or Form) menu pad.

You can mark a property or method as Public, Protected, or Hidden in both the New and Edit dialogs. Public is probably obvious—the property or method can be accessed by anything else. A hidden property or method is at the other end of the scale—it can't be accessed by anything except methods in the same class. A protected property or method is halfway in between—it can be accessed by subclasses of the class, but not by object instances or other objects.

Perhaps I should put that into plainer language. Suppose you have a form class, frmMaintME, that you use for creating simple maintenance forms. This form has an iidLastDeleted property, which contains the primary key of the last record that was deleted, and a WarnIfNotSaved method, which can be called, for example, if the user has attempted to navigate to another record without saving the changes to the current record. This method would display a dialog telling the user that they have modified a record but haven't saved it—and would they like to save it now?

If this property and method were public, they could be called from anywhere in the application, as long as the form instantiated from this form class was open. For example, you might create a form from this form class and add a custom command button to it. However, before allowing the user to click the button, you might want to test to make sure that changes have all been saved. You could call the WarnIfNotSaved method from the beginning of the command button's Click method.

You could even access those items from other forms. Generally it isn't practical to do so, but here's an example. Suppose, after deleting a record, you were going to ask the user whether or not to perform an operation with other records that included the ID of the deleted record. If you had to call another form to do so, you could reference the ID of the deleted record by accessing the property of the form from the second form.

Another example would be if you had stored a fully qualified file name in a property. You wouldn't want the user of the class to be able to enter garbage in this property, so you wouldn't mark it as public. Instead, you could mark it as protected. Then you would create a method of the class to enforce validation and other rules on the value that the user was entering; and only when the value passed would you allow the property to be assigned that value. Both the class as well as any subclasses of the class could address the property and work with it. The method would be accessed by the class user, not the property.

However, suppose further that this file name could actually be of two sorts—either a DOS-style filename like C:\WINDOWS\SYSTEM\MYAPP.DLL, or a UNC name like \\MAXIMILLION\DATA\SALES\REVENUES.DBF. You might not want any old subclass to be able to access this property, because the subclass might not know which type of file name it is. Instead, you could set this property to be hidden, and thus it would be hidden not only from forms instantiated from the class, but also from subclasses of the class. Those subclasses would still have to call methods of the class that were allowed to address the property.

This mechanism has a similarity to Control classes, in that you want to provide a very controlled interface for the user of the class; you don't want them calling any old method of the class, nor do you want them reading (or writing to) any old property. Instead, you make most of your properties and methods hidden or protected, and allow the user of the class to use just a few specific properties and methods that have been left public.

In this way, you can provide a class for others to use, and they can access the public and protected members according to certain rules. You've protected the internal workings of the class by making the appropriate properties and methods protected or hidden.

This is all getting a bit theoretical, so I'll stop now. It's good to be aware, however, that you have control over access to properties and methods of classes in a manner similar to the scoping (public, private and local) you can provide for variables.

## **The property and method hierarchy**

Now that you're comfortable with the idea of classes as well as how to use them, it's time to talk about the ramifications of this inheritance concept. As I've said earlier, I believe that the tag team of subclassing and inheritance is the key productive feature of Visual FoxPro's object-oriented capabilities.

## **Overriding property inheritance**

The definition of inheritance is that a subclass references the properties and methods of its parent class, and that parent class references the properties and methods of its parent, and so on up the line until the base class is reached. As a result, if the Visual FoxPro form base class has a white background color, every class subclassed from that base form will also have a white background, and so on down the line.

However, as we've also seen, if you create your form base class with a purple background color, every subclass created from your base class will also have a purple background color.

---

This is called *breaking inheritance*, and your base form's background color is said to have overridden the parent's background color property. (I'm using a property here as an example, but the same is true for methods, as you'll see in a minute.) If you continue to subclass several levels down, and, at level four you change the background color from purple to red, you've broken inheritance again, and all classes created from that red subclass will then have a red background color. Furthermore, of course, all forms created from the class with the red background color will have a red background. The important thing to know, now, is that if you change the background color of your base form from purple to green, the classes and forms that have red background colors will not change to green. Once you have overridden a property or method, you don't get it back automatically.

Remember that this is all property and method specific. Overriding one property in a class does not have any effect on any of the other properties or methods.

## Overriding method inheritance

Methods behave the same way, but there is additional functionality to be aware of due to the nature of program code vs. values. An object can't have more than one value of a property—for example, if a background color is blue, it can't also be yellow (no, the two properties wouldn't combine to create green). However, it is possible to "add to" a method.

Just as with properties, methods are inherited. Suppose you've created a command button with a Click method that contains the following code:

```
beep()
```

Thus, every time the command button is pressed, the computer's speaker beeps. Annoying, but suitable for this discussion. Now, when you subclass the command button, the subclass will also have a Click method, but if you open the subclass' Click method in the Code window, there won't be anything there. However, when you click on the command button, it will beep, because it inherits the Click method of its parent.

You'll want to be able to do two things. The first is to override the Click method code of the parent and replace it with another piece of code, such as the following:

```
wait window "You have pressed this Command Button"
```

You can see that the Click method in the parent has been overridden by the Click method of this class. If you subclass this class, that level will also execute the Wait Window code because we have broken inheritance, just as with the color of the forms above.

## Preventing a parent's method from firing

The next thing you're going to want to do is override the parent's method but not do anything in its place. Here's a nice trick. To do this, you needn't invent any complex code: Anything contained in the method of the current class will cause the parent's event to be ignored. As a result, you can use a simple comment, such as

```
* override
```

in the Click method of the form or subclass.

## Preventing an event from firing

Watch carefully, because this next point is subtle. The last several sections have all addressed the code in a method. As I discussed earlier, there is a difference between events and methods. There are times when you might want the behavior of an event to be suppressed. (Are you picturing the peasant in *Monty Python and the Holy Grail*, hollering, “He’s repressing me...”?)

For example, you can create your own pick list with a grid that features incremental search—as the user types a character, the highlight in the pick list repositions itself to the nearest entry containing those characters. The grid would have a text box in the column that’s being searched. As you know, the default behavior of the KeyPress event of a text box is to display the character typed in the text box. You don’t actually want the character typed to be displayed, so you would want that behavior—displaying the character—to be overridden.

Use the NODEFAULT keyword in the event’s method to do this. For example, you could write code to trap the character being typed, perhaps storing it to a property of the pick list, and then using NODEFAULT to prevent the character from being displayed, like so:

```
* they pressed an alphanumeric key
nodefault
* append the last key pressed to the keyboard buffer property
this.cStuffInKBBuffer = this.cStuffInKBBuffer + chr(nKeyCode)
* look for the complete string
if not seek(upper(this.cStuffInKBBuffer))
  go this.nCurRecNum
endif
```

The actual code would be more complex because you would want to test for the time elapsed between keypresses, but you get the idea.

## Calling a parent’s method in addition to the local method

Another task you’re likely want to perform is to call the parent’s method in addition to the code in the current method. In other words, this time you’re mixing the blue and the yellow colors, and want green. You can reference the code of the parent class using the :: operator (referred to as the *scope resolution operator*) or the DODEFAULT() keyword. The DODEFAULT() keyword is usually preferred because you don’t have to know the name of the parent class you’re referencing.

Suppose the parent class is named cmdDrill and you’ve subclassed it with a name of cmdDrillLocation. The click event of cmdDrill has the following code:

```
beep()
```

If you simply put code in the Click method of the cmdDrillLocation subclass, you would end up overriding any code in the cmdDrill class’s Click method. To call the BEEP code in the cmdDrill class as well as the code in the Click event of the cmdDrillLocation class, you would use the following code:

```
Dodefault()
```

---

```
<code in the cmdDrillLocation class>
```

or

```
cmdDrill::click()
<code in the cmdDrillLocation class>
```

Both of these code snippets would call the parent class's Click method first, and then execute the Click method of the current class. You could put a call to dodefault() at the beginning or end of a method, depending on when you wanted that behavior to be executed. Even better, you could put it in the middle of a method, and have the other code in the method act as a "wrapper" for the code in the parent class's method.

## Suggested base class modifications

Isn't it irritating to read lines like "Make modifications to your own form base class as desired," as I did earlier in this chapter? It's the post-college equivalent of "This exercise is left to the reader." When you were a student, you actually believed the author—but it really meant that the author had no clue as to the answer.

Similarly, wouldn't it be much more useful if I spent a couple more pages and described some concrete modifications for your own base classes? I know all the beta testers would have loved to have a series of examples when they first broke open their copy of VFP 3.0.

So here's a quick list of suggestions of properties and methods you might want to set or add to your classes. This is just to get you started—obviously you'll want to expand this list according to your own needs, wants, and style.

Control	Property/Method	Description
Everything	About (custom method)	Add to each class and use for your own extended documentation.
	Release()  Release This	
Forms	AutoCenter	Set to .T. so the form will display in the center of the screen.
	Caption	Change to "frm" so you know when you accidentally created a form with a VFP base class, because it will have a caption of "Form1."
	MaxButton	Set to .F. so the user can't maximize the form and then wonder why the form is full-screen but the controls are still jammed in the upper-left corner of the form.

<b>Control</b>	<b>Property/Method</b>	<b>Description</b>
Forms <i>(continued)</i>	Load()  Destroy()  *Hide the form so it *appears to go away *faster.  <code>This.Hide()</code>	See Listing 10.1 at the end of this table.
Labels	AutoSize = .T. BackStyle = Transparent Caption = "lbl"	
Command Buttons	Caption = "cmd"	
Page Frames	ActivePage = 0 PageCount = 0 TabStyle = Nonjustified	
All data-related controls—text boxes, check boxes, combos, lists, etc.	AnyChange (custom method)  Valid()  Validation (custom method)	Add the following code to the InterActiveChange and ProgrammaticChange events:  <code>This.AnyChange()</code>  Why? So you can centralize code that should be called if either the InterActiveChange event or the ProgrammaticChange event is called. You can still put code that is specific to either InterActiveChange or ProgrammaticChange in the respective method, but this technique saves a lot of redundant code.  See Listing 10.2 at the end of this table.  This method is called from the Valid() method.
Text and Edit Boxes	IntegralHeight = .T. SelectOnEntry = .T. Valid()	
Check Boxes	AutoSize = .T. BackStyle = Transparent Caption = "chk" Valid()	
Option Groups	OptionGroup AutoSize = .T. BackStyle = Transparent ButtonCount = 12 Valid()	Option groups have buttons that are not already subclassed and can't be subclassed. So ignore the Option Button class.  I create an Option group with a dozen option buttons that are already preconfigured. Then I set Enabled and Visible to False and resize the option group. That way, the buttons have the properties already.  Option button properties: AutoSize = .T. Caption = "1" through "12"
List Boxes	IntegralHeight = .T.	List boxes don't get a Valid() because the Valid() fires each time you move the highlight in the list box.

Control	Property/Method	Description
List and Combo Boxes	altems[1] (custom property) RowSourceType = 5 - Array RowSource = this.altems ItemTips = .T. SelectOnEntry = .T. Init() This.altems[1] = ""	Enter "altems[1]" in the property name so that Visual FoxPro knows it's an array.  You might want to turn ItemTips to True, so that the entire row is displayed when the combo is opened.  You might want to turn SelectOnEntry to True so that an item in the combo is automatically selected when the user tabs into the control.
Spinners	SelectOnEntry = .T.	
Grids	AllowHeaderSizing = .F. AllowRowSizing = .F. DeleteMark = .F. RecordMark = .F. SplitBar = .F.	

***Listing 10.1.** The Form controls' Load() method code from the above table.*

```
* Set some environmental things the way we want.
set talk off
if oApp.lCentIsOn
  set century on
else
  set century off
endif
set deleted on
if upper(oApp.cMethod) = "DEV"
  set escape on
else
  set escape off
endif
set exact off
set exclusive off
set fullpath on
set near off
set safety off
if oApp.cMethod = "DEV"
  set strictdate to 2
else
  set strictdate to 1
endif
set unique off
```

***Listing 10.2.** The data-related controls' Valid() method code from the above table.*

```
*\\\  
*\\\  
*\\\  
*\\\ except list boxes  
*\\\ compliments of Doug Hennig  
* If the Valid method is fired because the user clicked on a button  
* with the Cancel property set to .T. or if the button has an lCancel  
* property (which is part of the SFCommandButton base class) and it's  
* .T., don't bother doing the rest of the validation.
```

```
local oObject
oObject = sys(1270)
if lastkey() = 27 or (type('oObject.lCancel') = 'L' and ;
  oObject.lCancel)
  return .T.
endif lastkey() = 27 ...

* Do the custom validation (this allows the developer to put custom
* validation code into the Validation method rather than having to
* use code like the following in the Valid method:

* dodefault()
* custom code here
* nodefault

return This.Validation()
```

## Up close with the Class Designer

The Class Designer has the same interface and nearly the same functionality as the Form Designer. You can open it by using the MODIFY CLASS command or after you've defined a class with the New Class dialog that results from the CREATE CLASS command.

The differences from the Form Designer have to do with the fact that a class is a definition of an object and can't be "run" itself, any more than the mold for a firing pin can be placed into a rifle and used to fire the cartridge. Thus, the Run icon is disabled when the Class Designer window is active, and the shortcut menu does not have a Run menu option.

The Class menu pad appears on the menu bar when the Class Designer is the active window. The first four menu options—New Property, New Method, Edit Property/Method, and Include File—are the same as those found on the Form menu that appears when the Form Designer is the active window.

The fifth menu option, Class Info, opens the Class Info dialog. This dialog is used to attach information and icons to the class.

The Toolbar Icon text box and ellipsis button allow you to select an icon that will be displayed on the class's toolbar button when you register this class under the Tools, Options menu option, and beside the name of the class when it is indented under the class library name in the Project Manager. The Container Icon text box and ellipsis button allow you to select an icon that will be displayed in the Class Browser.

The OLE Public check box is used to specify that the class can be used to generate a custom Automation server when it is built through the Project Manager.

The Members tab of the Class Info dialog simply has a list box that displays all members of the class. A "member" is a generic term that encompasses both properties and methods. This list box isn't all that useful, actually, since (1) it doesn't identify whether a member is a property or a method (yeah, it *should* be obvious, but if you're spelunking through someone else's class, it may well not be), and (2) it doesn't really tell you much. The Edit Property/Method dialog is much more useful in my opinion. Click the Modify button on this tab to get to the Edit dialog.

## Up close with the Class Browser

The Class Browser is a tool that helps in managing class libraries. Because multiple classes can be contained in the same class library, even to the extent of multiple levels of classes, it can be difficult to keep straight where in the class hierarchy a specific class is located. Furthermore, because classes reference each other through file name references stored in the class library, it can be very difficult to perform simple maintenance tasks such as renaming a class or moving it from one location to another. The Class Browser makes short work of many of these tasks.

### Starting the Class Browser

You can select Class Browser from the Tools menu or call it manually using the following line of code:

```
do home() + "browser.app"
```

If you do so, you will be prompted for the name of the class library with which you want to work.

You can also store the name of the Class Browser application, BROWSER.APP, to the \_BROWSER system memory variable (in the File Locations tab of the Tools, Options dialog), and then issue the command:

```
do (_browser)
```

The Class Browser takes an optional parameter. If you pass it the name of a class library, you will not be prompted for the class library:

```
do home() + "browser.app" with "baseform"
```

### Using the Class Browser

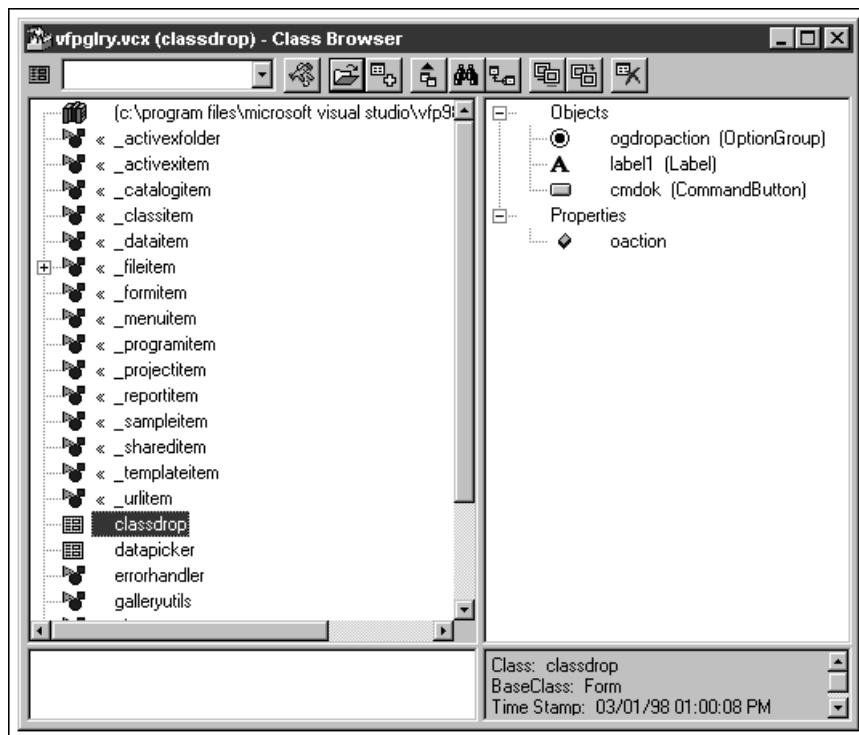
The Class Browser is a single form that displays all classes in a class library. It has a number of functions that allow you to view and manipulate those classes in a variety of ways. See **Figure 10.9**.

The list box in the upper-left quadrant of the Class Browser shows the hierarchy of the classes in the class library you are browsing. To view a single class type, you can filter the list on the left by using the Type drop-down combo box.

The right panel contains a list of all exposed members of the selected class. You can right-click and select to display Protected, Hidden, and Empty members as well. The Class Browser will remember the settings you've selected from session to session.

Here is a list of toolbar buttons, from left to right:

- Component Gallery—Opens the Visual FoxPro Component Gallery, which will be discussed in Chapter 17.
- Open—Opens a class library (and closes the current one, if one is open). If you right-click on the button, you'll see a list of the class libraries you most recently opened.



**Figure 10.9.** The Visual FoxPro 6.0 Class Browser.

- View Additional File—Opens another class library in the Class Browser. This is particularly handy if your inheritance hierarchy spans multiple class libraries. Again, right-clicking displays a list of libraries you recently opened.
- View Class Code—Generates a file called VIEWCODE.PRG, which is the programmatic equivalent of the class.
- Search—Allows you to search through a class or class library for a string of text. Note that because you can search for text in the descriptions as well, it behooves you to document your classes well!
- New Class—Creates a new class in the same fashion as the CREATE CLASS command. However, the Based On and Store In values are already filled in with defaults pointing to the current class and library in the Class Browser.
- Rename—Allows you to rename a class or properties and methods. Renaming a class can be very dangerous because it can break inheritance. If you rename a parent of a class, you might not be able to instantiate that class anymore—nor will you be able to even open the class to modify it. You might as well throw it away. Renaming properties and methods can be equally dangerous, because source code in other methods that referenced the old name will no longer work.

- **Redefine**—Allows you to change the parent class of a class. Suppose you had two text-box classes, `txtEditable` and `txtReadOnly`. You then subclassed `txtReadOnly` to create a specialized data-entry text box, and only after you were just about done did you realize that you had subclassed the wrong class—you actually meant to subclass `txtEditable`. (How useful would a `ReadOnly` data-entry text box be?) **Redefine** allows you to fix this type of mistake quickly and easily. You need to make sure that the target class has the same structure as the source class—if it doesn’t, you can lose code and otherwise mess up your classes.
- **Clean Up**—Essentially performs a “pack” on a .VCX file. When you modify a class library, the records themselves in the .VCX aren’t modified; they’re deleted and new copies are appended. As you know, the deleted records are still hanging around—they’re just flagged as deleted. As a result, a .VCX undergoing a lot of maintenance can get very large. This function will remove all of the deleted records permanently.

The icon to the left of the combo box represents the class selected in the list box. You can use this icon to place an object of that class on a form—yes, yet another way to add controls to a form! Drag the icon onto the Visual FoxPro screen or onto an appropriate container (like a form).

You can also modify a class from the Class Browser by double-clicking on the class in the list box.

Chevrons next to a class in the list box indicate that the parent class for that class is not in the same class library and isn’t currently open in the Class Browser.

The Class Browser was designed with an open architecture so that others could extend it as needed. One method is to write a separate application and register it with the Class Browser. These add-ins can be displayed by clicking the Add-ins button.

There is a world of hidden functionality in the Class Browser. Markus Egger has documented just about all of it in detail in his book, *Advanced Object-Oriented Programming with Visual FoxPro 6.0*. If you’re at all interested in using the Class Browser, I suggest you refer to this book.



# Chapter 11

## Output: The Report and Label Designers

I wish I had a nickel for every time I've repeated the adage that people generally don't put data into an application simply to have it there—they eventually want it back out at some point. And that's the topic of this chapter: getting data back out of a database. More specifically, using Visual FoxPro's Report and Label Designers.

The Report Designer and Label Designer are Visual FoxPro's tools for creating output and sending it to the screen, printer, or a file. As with the Menu Designer, you can automate part of the creation process by using the Quick Report feature. Furthermore, Visual FoxPro contains an additional pair of tools, the Report Wizard and the Label Wizard, that can design and produce a wide range of reports and labels. In fact, upon investigating the Wizards, you might feel you could skip the rest of this chapter; depending upon your specific needs, you might well be able to. Nonetheless, I'll discuss the Report Designer and Label Designer in the same depth as the other Visual FoxPro tools, so if you need to create more complex output, you'll have the wherewithal to do so.

The Report Designer and Label Designer work the same way in most instances, so I'll focus on the Report Designer for the bulk of this chapter, and then mention differences or additional capabilities germane to the Label Designer where appropriate.

### Quick start to building reports

In most of the other chapters in this section, I've taken a step-by-step approach, showing you how to quickly and easily build a simple widget of one sort or another, and then getting into the nuts and bolts of the tool once you've seen what the final product should look like. This "quick start" will be slightly different, because there are two different philosophies regarding the creation of a report. I'll discuss each position and why I think one is superior, and then I'll explain how to proceed along those lines.

### Report terminology

Before I get into building reports, I should discuss their key terms and how they are structured.

The simplest report is a tabular listing that looks like a Browse window—multiple rows with one or more columns that don't even fill a page. The data can come from one table or multiple tables; for the time being, I'll assume there's no difference. Each time a "row" from the Browse window prints, it's called a *detail row*. Unlike a Browse window, however, you can print data from a row on more than one line—which is handy if you've got so much data that you can't fit it on a single line.

If the report spills onto multiple pages, you're probably going to want to display some information on every page. If this information is at the top of the page, it's called a *header*; if

it's at the bottom, it's called a *footer*. This information might be "dumb" data, such as column headings, the date of the report, or the page number. But it might also be intelligent, such as a running total or other calculated values.

Once your report spans enough pages, you might find that you want a cover for your report, just like the fancy report you did for your 11th-grade civics class. You might also want a page at the end that summarizes the information in the report. These pages are called *title* and *summary* pages, and they can be printed as completely separate pages for your report. In some cases, however, you might find that while you need some information printed only once at the beginning or end of your report, you don't need it on a completely separate page.

Having a long report with pages and pages of data isn't always as useful as it could be by itself—often you'll want to break down the report into logical groupings. For example, in a report of club members, you might want to go beyond simply alphabetizing the members, and provide a break each time the first letter of the last name changes. Or perhaps you want to categorize members according to some criteria: "Full-Time" members in one group, "Associate Members" in a second, "Junior Members" in a third, and "Out-Of-State Members" in yet a fourth group. Within each of these groups, you would still alphabetize them by name, of course.

Each of these groups is called, funny enough, a *group*, and can simply print one after another, or each group can start on its own page. Furthermore, you can print some sort of information at the beginning of the group or at the end. For example, you might want to print the name of the group along with some descriptive information at the beginning of the group, and then print the total number of members of the group along with some other calculated information, such as total donations for the year, at the end of the group listing. These sections would be called *group headers* and *group footers*, and they'd act similarly to page headers and footers, except, of course, they print only with respect to a group.

There can be one or more levels of groups—imagine if the aforementioned club directory was grouped first by state of residence, and then by Member Type within each state.

## To-may-to or tuh-mah-to?

In the client-server world, there's an age-old discussion regarding whether to put validation rules on the server or in the user interface. The argument for server-based validation revolves around the idea that it's a centralized, foolproof place for rules that will prevent garbage from getting into the database. The opposite position is based on the theory that by the time bad data gets to the server to be validated, it's too late to send useful messages back to the user. I won't argue this conundrum here, but I offer it to serve as a reference point for another argument: how to structure and process reports.

A typical report writer, Visual FoxPro's included, contains much more functionality than that which is simply needed for printing information from a table. Indeed, not only can one format data, set attributes for fields regarding whether and when to print, but one also can do calculations and logical manipulations of data according to the location on the report and its relation to other data. You can even write code that will fire off at certain times during the processing of a report!

The argument, then, revolves around whether or not to use all of this functionality. The justification for doing so is that it provides a great deal of power and flexibility, often fairly easily. The downside, the naysayers argue, is that testing and debugging, as well as future maintenance and modification, become significantly harder. And while it's not much fun

---

debugging a problem anywhere in an application, trying to figure out why the group subtotal on page 4 is always off by a nickel is as unrewarding as any debugging task a programmer can face.

## Dumbing down the Report Writer

I can hear you asking, “So if you’re not going to perform all sorts of fancy tricks with the Report Writer, what’s the alternative?” Here’s the right way, in my not-so-humble opinion, to build reports for use in Visual FoxPro.

First, build a denormalized cursor that represents the entire data set you want to print—grouped and sorted as it should print out. Then use the Report Writer to simply print that report, much like you use a Mask property to display a value in a certain way. No calculations, no tricks, no fancy methods firing here and there.

It isn’t always exactly this easy; if you’re trying to print images, or you need to format values in specific ways, you’ll probably have to get into the Report Writer to some extent. But use those requirements as the exception, not the rule.

Why am I so adamant about this? Because, as with everything computer-related, requirements always become more demanding. When was the last time you had a user or a customer ask to “get rid of a bunch of functionality”? When did they try to make your life simpler? That’s right, requirements always get more demanding. Even the simplest report ends up having this gizmo and that widget added on, and suddenly what was fairly straightforward ends up looking like a contraption from a mad scientist’s laboratory.

The problem with this is that there isn’t any easy way to document any of the work done in the Report Writer itself. Suppose you create a complex expression that prints one value in one situation, a second value another time, and yet a third value if two other conditions are met. If you were displaying this value on a form, you’d be able to write comments somewhere in the form that describe what you did, how you did it, and why. But there’s no place in a report to do that.

As a result, if you do all the processing of the report’s data in code, you can document the same decisions and explanations, and then simply print the results. When the report comes out wrong, you can take a snapshot of the cursor and debug your code—which you know how to do. You don’t have to delve into some arcane magic in the Report Writer that you put together months ago.

Furthermore, when it comes time to output that same data to another format—perhaps a different report writer that has additional functionality, or to a different output target, such as a spreadsheet or another database file format—you’ll have virtually all of the data in place. If you had done a lot of the calculations and logical decisions in the report form itself, you’d end up rewriting all that again so you could send the results elsewhere.

## How about an example?

This might be one of those places where an example would help visualize what I’m talking about. Suppose you’re using the Report Writer to print an invoice. You could set up a series of relations between the Customer, Invoice Header, Invoice Detail, Product, and Vendor tables, and then roll through this set of joins. Problems arise when there isn’t a match in one of the joins—for example, if one of the Products identified in the Invoice Detail is missing from the

Product table. Or when the calculated subtotals for the Product Lines on the invoice don't match the grand total. Or there might be a set of complex rules about when certain attributes of specific products get printed. Perhaps a certain item gets taxed in one jurisdiction, but not in another. Certain Product Lines require footnotes or installation remarks. Distributors require unit and extended prices, while walk-in customers just get extended. And so on ... you've all run into those rules that seem designed just to make your life as a programmer tougher.

Instead of coding the logic for each of these rules into the report form—where they have to be maintained, and duplicated when another similar invoice form is designed—put the logic in the code that generates the cursor that feeds the report.

First, create a SQL SELECT statement that creates a row for each item detail in the invoice (or set of invoices). In this row, you'll also have the data that goes into the Invoice Header, including Invoice Number, Invoice Date, and Customer Purchase Order Number, but also Invoice Header data from other tables, such as Customer Name, Billing Address, and so on. Because the Invoice Detail record most likely had a foreign key to the Product table (which in turn had a foreign key that pointed to the Vendor table), you'll need to grab the appropriate data, such as Vendor Name (or at least Abbreviation), Product Number (and Description), and perhaps a Gross Unit Price for the item as well.

You can do calculations in this SELECT, extending purchased quantities into extended totals based on Gross and/or Net Prices. At this point, you might calculate several extended prices, based on different conditions such as whether or not the customer pays tax, freight, installation costs, or a variety of other factors.

For example, suppose that, when ordering, the customer gets 10% off a specific item if they present a certain coupon or recite a jingle from the local soft-rock radio station, and they get 15% off if they do both (okay, that's kind of a silly promotion, but then, think about it...). Whether or not they do so is recorded in a logical field in the Invoice Detail table. When calculating the Invoice, you could have a Unit Price field on the invoice report that contained the expression:

```
nAmtUnit * iif(1SangJingle and 1HadCoupon, ;  
              0.85, ;  
              iif(1SangJingle or 1HadCoupon, 0.90, 1.00) ;  
              )
```

But, well, blech. What if business conditions changed, or (more likely) management changed its mind and suddenly you had to change this rule? Or what if your invoices weren't cross-footing properly? Trying to debug this expression while running the report would be a real pain.

Better, then, to include the following columns in your SQL SELECT statement:

```
nAmtUnit  
1SangJingle  
1HadCoupon  
nMultiplierPercentage  
nAmtDiscount
```

where:

---

```
nMultiplierPercentage = ;
  iif(lSangJingle and lHadCoupon, ;
    0.85, ;
      iif(lSangJingle or lHadCoupon, 0.90, 1.00);
  )
```

and then:

```
nAmtDiscount = nAmtUnit * nMultiplierPercentage
```

The `nMultiplierPercentage` value, instead of being stored deep inside the Report Writer, is in the code, where it's easy to find and debug.

If you ran into problems using the first method, you'd have to test for (1) bad data in the tables, (2) a bad relation joining some of the tables, (3) a bad formula in the report, or (4) some other hidden factor. Because all of the relevant data is in the result set gathered by the SQL SELECT statement, you'd have a much easier time debugging. And because you're gathering only a small set of data, you don't have to worry about performance—the result set is typically only going to be a handful of records, or a few hundred at most.

You can make multiple passes against this result set if you like. For example, what if you needed subtotals? Those are pretty easy to do in the Report Writer, as you'll see shortly. But often subtotals are needed not only after the end of the group, but during printing—and sometimes before the group is printed.

For example, the stated format of the invoice requires including the Product Line subtotal at the beginning of the group of items for that Product Line. This is impossible to do under normal circumstances, but if you've included a `SUM()` expression in your `SELECT`, the subtotal is available there for you as you begin running the report.

For more complex requirements, you can simply SCAN through the records in the result set and calculate additional data on the fly or stuff empty columns that you put in the `SELECT` for just that purpose. Because you've got a paucity of records in the result set, Visual FoxPro will handle this type of processing in the blink of an eye. Then again, you simply have to print out pre-calculated data and, as I mentioned earlier, it's much easier to make changes and debug.

An additional advantage to this mechanism is that you can include non-data-source information in your report quickly and easily. For example, suppose the user entered some data, such as a user-specified report heading, into a form that launched the report. You can capture this data from the form, stuff it into the cursor, and then print that information as needed just as if it were another data field. Normally, you'd have to mess around with report variables or some other nonsense, worrying about initializing it, making the value change when needed, scoping the variable properly so it's visible to the report, as well as the standard headaches involving debugging and future maintenance. You don't need all that.

## Up close with the Report Designer

Using the Report Designer involves a number of tools: the Report Designer window itself, several toolbars, the Data Environment window, and the Properties window. In addition, the menu changes according to which window is active and which function is being executed. I'll examine them one by one.

### The Report Designer window

You can open the Report Designer in one of several ways:

- Open the File menu and click New, Report, and then New File.
- Enter the command CREATE REPORT in the Command window.
- Select the Reports object in the Project Manager window and then click the New button.

In any case, the Report Designer window appears as shown in **Figure 11.1**.



**Figure 11.1.** An empty Report Designer window.

The empty Report Designer window features three bands by default, each identified by a gray bar underneath the band itself. The most critical concept to understand about reporting is how these bands work. In succinct terms, a report form is run against a table, and the report form, in its entirety, is evaluated as each record in the table is processed. You place objects in each band, and then those contents are printed at specific times and for specific reasons according to the type of band and attributes set for the object.

Mastering the rules of “which band is printed when” will enable you to use the Report Designer to produce, in concert with the appropriate underlying table, any type of report you require. In addition to the three bands you see in an empty report, there are four more: the Group Header and Footer, and the Title and Summary bands. In short, here are the rules for each band:

- The contents of the Detail band are printed once for each record in the underlying table.

This means you can place several rows of fields in the Detail band, and they will all be printed for each row in the table that is driving the report. In fact, you could make the Detail band the size of a full sheet of paper, and thus end up printing one page for each record. This would be handy if you wanted to print invoices, for example, or membership renewal forms. You would simply have one record for each invoice or member, and then print a “report”—instead of laboriously hand-coding a full-page document like in the old days.

- The contents of the Page Header band are printed once each time a new page is started.
- The contents of the Page Footer band are printed once each time a page is completed.

Most often, these bands are used to carry information from one page to the next, such as page numbers, dates, and report headings. But you can also use them to print column headings or column footers. With a little bit of ingenuity, you could even print fields that change according to the data on the page, such as the first and last name on a page in a directory.

- The contents of the Group Header band are printed each time the expression upon which the group is based changes.
- The contents of the Group Footer band are printed each time the group of records that belongs to a group has finished printing.

Read the wording of these two sections very carefully. When you use groups in a report, it’s not enough to simply create a group expression and its associated band. You’ll also need to sort the report’s data set so it’s in order according to the group expressions. For example, if you were grouping a sales report by Sales Region, and then by Sales Rep within Sales Region, you would want to make sure the results were sorted on Sales Region, and then by Sales Rep within Sales Region.

- The contents of the Title band print at the very beginning of the report.
- The contents of the Summary band print at the very end of the report.

Once you’ve created a report (I’m assuming you’ve created a report or two with the Wizard in your spare time), it’s likely that you’re going to want to change the contents of the various bands. The contents are called *controls*, although this term includes any object that can be placed on a report, including text labels and boxes. The following sections describe how to perform the most common tasks.

### Select a control

Click on the control so the eight black sizing boxes at the corners and in the middle of each side appear.

**Select a group of controls**

Click on a control and then, while holding down the Shift key, select additional controls; or position the mouse pointer outside all controls, and then click and drag the mouse pointer so that it at least partially overlaps the controls desired. The dotted line that appears is called a *marquee* (because it flashes like the lights that surround a movie sign). You can then deselect individual controls by holding down the Shift key and clicking on the control in question. Once you've selected a group of controls (by using either method), most actions that can be performed on a single control (such as moving, duplicating, resizing, and changing font) can also be performed on the group.

**Move a control within a report**

Select and drag the control to its desired location. This works across bands as well as within a band. You can also use the cursor keys to nudge a control a pixel or two when dragging doesn't provide discrete enough resolution.

**Move a control from one report to another**

Select the control and use Edit, Cut to remove it from the original report. Select the second report and use Edit, Paste to place the control on it.

**Duplicate a control**

Select the control and use Edit, Copy and Edit, Paste (or the Ctrl+C and Ctrl+V key combinations) to make a copy. Then move the new copy of the control to its desired location.

**Resize a control**

Select the control and use the sizing handles to resize it as desired. You can also use Shift and a cursor key to resize by small amounts when dragging doesn't provide discrete enough resolution.

**Delete a control**

Select the control and press the Delete key. Note that Delete does not place the control on the clipboard like Cut and Copy do. However, you can recover a control erroneously deleted (or a group of controls deleted in one step) by selecting the Edit, Undo menu option.

**Change the font or other attributes of a control**

Select the control and use the appropriate menu option in the Format menu. The attributes available depend on which type of control is selected. For example, you can't change the Font of a line.

**Resize a band**

Place the mouse in the gray bar underneath the band, and drag it in the direction to be sized. If you can't make a band smaller by dragging, a control in the band is in the way. The control might not be visible if the width of the report form is wider than the current window, if the control is a single line with no thickness (hard to see), or if the control is the same color as the report background (impossible to see). To hunt these down, expand the band and then try to

---

lasso hidden controls starting around the edges of the band to “scoop up” open areas to find the bothersome controls.

## The Report Controls toolbar

So far I’ve shown you how to change the external attributes of one or more controls, but I’ve not said anything about modifying the internal properties of a control—the contents of a field or the text in a label. As with the Form Designer, Visual FoxPro has provided a toolbar to facilitate these tasks. Select the Report Controls toolbar from the View menu to display it.

There are six types of controls that can be placed on a report form. These are (in order of appearance on the toolbar) labels (simple strings of text), fields and expressions, lines, boxes, circles and ovals, and pictures. You select the appropriate icon from the toolbar to create it. With the exception of text, you can modify a control just by double-clicking on it.

The following sections describe how to perform the most common tasks using the Report Controls toolbar.

### Change the text of a label

Select the text icon (the letter A) and then click on the label. The cursor will change to an “insertion point” much like in a word processor. You can then type text (in insert or overwrite mode), backspace, and delete.

### Change the internal properties of a control

Select the control and either double-click or right-click and select the Properties menu option from the context menu. In either case, the resulting dialog will allow you to edit the internal properties specific to that control.

### Create a control

Select the appropriate icon for the type of control to be created and then draw the control on the report form (select a point and drag the mouse until the control is laid out as desired). If you’re creating a Field control, the Properties dialog will appear so you can specify the expression and other information. You’ll have to bring up the Properties dialog for other controls yourself.

Notice that the icon you select when creating a control “pops out” once you’ve created the control. If you are going to create more than one of the same type of control, this is a nuisance. You can keep the icon “pushed in” by selecting the last icon in the toolbar—the button lock—or by double-clicking on the icon. Release the button lock by clicking on it again or by selecting the arrow icon. Selecting the arrow icon also will release any other icon that was pushed by mistake.

## The Layout Controls toolbar

As you work with controls inside a report, you’ll find yourself repeating some tasks over and over. Visual FoxPro has shortcuts to a number of these common tasks, including aligning and resizing controls. These are located on the Layout toolbar, which is accessible from the View menu. (This is the same toolbar used with the Form Designer.)

The first four icons allow you to align a group of controls with the top, bottom, left, or right edges; the next two align controls according to their horizontal or vertical centers. The

next three icons allow you to make all of the controls the same height, width, or height and width. The next two icons center a control either horizontally or vertically (used together, they will center the control both horizontally and vertically). The last two allow you to change the relative position of overlapping controls (pulling one from behind to the front or vice versa).

## **The Color Palette toolbar**

You can change the foreground and background colors of controls by using the Color Palette toolbar, which is also available from the View menu.

To change the foreground or background colors of a control, select the control, select the foreground or background icon in the Color Palette toolbar, and then select the color. You can select additional fixed colors, as well as create and assign custom colors with the Other Colors command button in the Color Palette toolbar.

## **The Report menu**

The Report menu pad is available only when the Report Designer or Label Designer window is the active window.

The Title/Summary menu option is used to create Title and/or Summary bands for a report. The Title band is printed once at the beginning of the report; the Summary band is printed once at the end of the report. Select the appropriate check box to create the band, and then place controls in the band. You can force the Title and/or Summary bands to print on separate pages by selecting the appropriate New Page check box. Unchecking either the Title or Summary band check box will delete the band from the report—even if there are controls in the band—without first giving you a warning message.

The Data Grouping menu option is used to create group bands for a report. You can create up to 20 levels of groups. The Grouping Expression mover control allows you to create and reorder multiple group expressions. You can either type an expression or use the ellipsis command button to open the Expression Builder. Groups are numbered in the report form in the order they were created, but you can change the order by dragging the expressions to the desired locations with the mover buttons on the left side of the mover control.

The Group properties box within the Data Grouping dialog, as shown in **Figure 11.2**, allows you to control how each group appears on the report.

If you are creating a multicolumn report, you can force the start of a new column when the group changes by selecting the “Start group on new column” check box. For all reports, you can force a group to start on a new page by selecting “Start each group on a new page” check box. You can also start page numbering with 1 each time a new group is encountered by selecting “Reset page number to 1 for each group” check box. Note that the “Start each group on a new page” check box is automatically checked and cannot be unchecked if you select the “Reset page number” check box. You can also have the Group Header band reprinted on each page (regardless of whether “Start each group on a new page” is checked) by selecting the “Reprint group header on each page” check box.

Finally, the “Start group on new page when less than” prompt enables you to control when to begin a group on a new page. If the Group Header is printed at the bottom of a page, most or all of the group’s records may print on the following page (this works like widow control in a word processor). To force the Group Header to that page as well, adjust the setting of the

spinner to an appropriate value. You can experiment with values made up of the height of the Group Header plus a multiplier of the height of the Detail band. For instance, if the Group Header is 2 inches high and a Detail band is 0.5 inches high, setting the spinner value to 2.9 will force the Group Header to start on the next page if fewer than two Detail records will fit on the remainder of the page.



**Figure 11.2.** The Data Grouping dialog allows you to add, edit, and delete group bands in a report.

The Variables menu option allows you to create memory variables that are evaluated while the report is running. These variables are available only during the execution of the report and are generally not visible to the rest of the application. A report can reference regular memory variables defined as public or private outside of the report as well, of course, just as it can reference field expressions. Report variables are handy for situations such as calculating rolling averages, counting through multiple groups, or calculating “percent of total” expressions that are evaluated for each record.

The Default Font menu option allows you to set the default font for the report form. This font is used when a Quick Report is created, as well as when additional controls are placed on the form. Note that the default font does not modify existing controls. (You would use the Format, Font menu option to change the font of existing controls.)

Clicking the Private Data Session menu option will set the report to have a data session that does not change if you open or use tables in other designers. When you add tables or views to your report’s data environment, you are changing the data sources that supply the report. It is

possible to make changes to the global data session with other designers (such as the Form Designer), and these changes can impact the data session of the report you are creating. Setting the data session for the report to Private will prevent this from happening. Of course, doing this prevents you from being able to create a cursor before running the report.

The Quick Report menu option allows you to automatically place fields and related controls on an empty form using one or more open tables as data sources.

The Run Report menu option allows you to print the report. It uses the standard Print dialog.

## **The View menu**

A number of menu options are added to the View menu when the Report Designer window is active. Depending on whether or not a report is being previewed, the Design or Preview menu option is checked. To switch to the other view, select the other menu option.

The existing report toolbars are removed and the Preview toolbar is automatically displayed during Preview. The original toolbars are restored when returning to Design view.

The Preview toolbar allows you to move from page to page, move to a specific page, change the zoom factor, print the report, and return to Design view.

You can create a data environment that is set up with the report. If you have a data environment attached to a report, it handles the tables—opening and closing them, setting orders and relations, and so on—just like with the Form Designer. You don't need to manually open tables, set orders and relations, and so on. Simply opening the report will also open up the data environment.

Three toolbar options are automatically added to the View menu when the Report Designer window is active: Report Controls, Layout, and Color Palette. When a toolbar is visible, its menu option is checked. The toolbars you select will automatically be opened the next time you open the Report Designer.

You can make grid lines visible on the Report Designer window by using the View, Grid Lines menu option. You can use the Format, Snap to Grid and Format, Set Grid Scale menu options to control how the grid lines are used. Snap to Grid forces objects to align to the grid lines on the Report Designer, while Set Grid Scale allows you to control the spacing of grid lines and what measurement is used.

The Show Position menu option displays in the status bar the dimensions and location of the currently selected control in the Report Designer window (or the combined dimensions and locations if multiple controls are selected), or the location of the cursor if no controls are selected. The measurements used are determined by the choice selected in the Format, Set Grid Scale dialog.

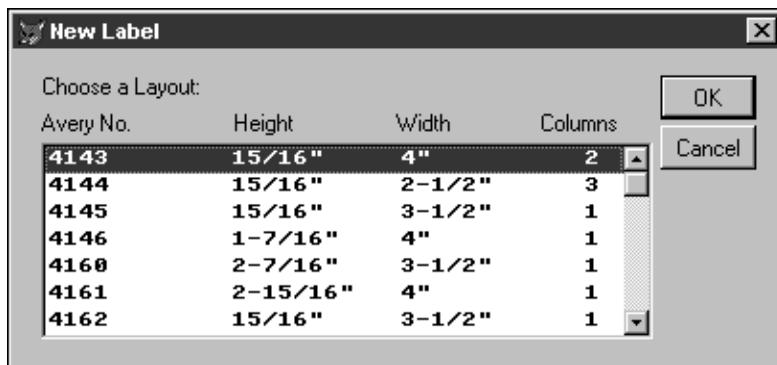
Once you've created a report, you'll notice that its size has automatically been set to 8.5 x 11. How do you get a landscape report instead? The trick is to avoid the View and Report menus—open the File, Page Setup menu option, click the Print Setup button, and choose the Landscape option button.

## Up close with the Label Designer

Working with labels, and with the Label Designer in particular, is very similar to working with reports and the Report Designer as far as manipulating expressions, creating bands, previewing and printing, and so on. Here I'll cover a few additional capabilities peculiar to this process.

First, when you create a label, you might want to use a label template that defines the size and style of label: how wide, how tall, how many across on a page, and what that page looks like. Visual FoxPro comes with a set of default label sizes (based on labels available from Avery, a leading label vendor), but, unfortunately, they are not installed automatically. If you attempt to CREATE LABEL and they're not installed on your computer, you'll get a message stating that there are no default labels installed. Of course, as with all other Microsoft error messages, there's no mention about what to do about this situation.

To install the default label templates, you can simply run the Label Wizard by opening the Tools menu and clicking Wizards, then Labels. After you pick a data source in step 1, you'll be prompted for a template in step 2. If Visual FoxPro doesn't find any existing templates, it will ask you whether or not you want them to be installed. Answer yes, and then cancel out of the wizard. You'll now have the templates installed, and when you issue the CREATE LABEL command, you'll be prompted with a dialog that allows you to select a template, as shown in **Figure 11.3**.



**Figure 11.3.** Visual FoxPro will ask you for a label layout when you create a label manually.

There are enough tricks with creating labels that you may want to consider using the Label Wizard to create all of your labels, and then just tweak the results with the Label Designer as needed.

## The Label Designer window

Creating a label is different in several ways. First, when creating a label (either through the File, New, Label menu option or by typing the CREATE LABEL XXX command), you'll be asked to choose a label size. (These label sizes are stored in the FOXUSER resource file.) Next, if you use Quick Report you'll notice that the Titles check box is not checked, because you usually

won't want the titles of the fields to appear on a label. Third, the form layout is selected by default, because, again, that's probably what you are going to want.

Note that you can create any kind of band for a label. You might want to have a cover page for the labels and use the Title band for this, or print a Summary page of the number of labels printed and their types. You could also, depending on the type of label used, print a header at the top of each page (on the edge of the label form) for control or audit purposes.

In the Label Designer window itself, you'll notice a new kind of band—the Column Header and Footer. Because labels are by default a columnar type of report (even if you have only one column on a page), you can place controls on the top and bottom of each column, instead of having to duplicate the same controls for each of the two, three, or four columns on the page.

## Tips and tricks

You can create a multi-column report by selecting the File, Page Layout menu option and changing the number of columns with the Column Number spinner. When you do so, the Report Designer changes to display only one column, and the Column Header and Footer bands appear as they do with the Label Designer.

You can control the data sources for a report in one of two ways. You can use a data environment stored with the report as I did with one example above, or you can specify a particular data source when you run the report. The second situation is handy when you are using the same report across multiple sets of data.

If you are going to be using multiple sets of data to populate the report, be sure to uncheck the Alias check box in the Quick Report dialog if these data sets contain tables with dissimilar names. This would happen if you have multiple tables with the same structures but different names. For example, if you have tables from multiple sales regions and they are all named according to their region, but they all have the same structure, you'll want to leave the field names unaliased.

Lines, rectangles, circles, and pictures are static controls. In other words, they don't change as the report processes records in the data source. However, they can be controlled by logical expressions, such as "print this box only if the amount is greater than zero."

When creating a Title Page or Summary Page, use the Layout Controls to center the data that goes on the page. Note that centering works only within the size of the band defined in the Report Designer. For example, if the height of the Title band is 5 inches and you create a separate title page and then center a title vertically, the title will display at 2.5 inches, not at 5.5 inches.

You don't have to use the Data Grouping feature to prevent a field from repeating. Suppose you're printing the PERSON table and have it ordered by Title, but don't want "President" and "Vice President" to print over and over. On the other hand, you don't want to group the report on Title, either. So open the Report Expression dialog, open the Print When dialog, and then set the Print Repeated Values option button to No.

## Is that all there is?

After you've worked a bit with the Report and Label Designers, you'll undoubtedly feel shortchanged when comparing their functionality with the richness found elsewhere in Visual FoxPro. For example, the Report Writer has a number of deficiencies that haven't been corrected in over half a decade. And you might also be wondering why these designers aren't object-oriented like other parts of VFP.

Unfortunately, I can only sing the same sad song that I did with the Menu Designer. Even worse, the rationale that I used with the Menu Designer doesn't quite cut it with reports. You can get by with a five- or six-year-old Menu Designer without much problem, but the limitations of the Report and Label Designers are significant shortcomings when it comes to building applications.

The bad news is that it's not going to get any better. Microsoft has been fairly blunt in explaining that these tools are simply not going to be enhanced. Instead, you are expected to either "make do" with what you have, or use other tools in their place. For example, Microsoft points you to third-party report tools, such as Crystal Reports, when you ask about specific shortcomings. And as you'll soon see, Microsoft intends to place Visual FoxPro in the middle tier of the three-tier architecture they've been pushing over the past few years. As a middle tier, Visual FoxPro has no use for an interface, and thus no need for a menu, nor any need for output, and thus the Report Writer is at a dead end.



## Section III

# Building Your First Application

Now that you're done with Section II, you've got all the tools you need to use Visual FoxPro interactively and programmatically. It's time to put it all together and build an application. In this book, I'll show you how to build three types of apps: the traditional LAN app; a two-tier client-server app where VFP is the front end, talking to a back-end database; and a multi-tier app with VFP as the middle tier. The last two types will be "lite" versions of the LAN app, and—due to space and time constraints—are included only in the .CHM file.

To show you how to get started, I'll use a LAN application because it's the type of system you're probably most comfortable with. Once you understand how to use all of the tools in concert, and how object orientation works in VFP, you'll have the foundation needed to venture on to the two-tier and three-tier systems.



# Chapter 12

## The Structure of a LAN Application

By now you've probably got a variety of files that you've played around with in various test directories: programs, forms, menus, class libraries, maybe even a report or two. But now that you're comfortable using each individual tool, it's time to learn how to build an application. There are generally three types of applications you can build with Visual FoxPro. The first is the "monolithic" LAN app that we all know and love, which is built with VFP from head to toe. The second is a two-tier (or, often, "client-server") app where the user interface is separated from the back-end data, and Visual FoxPro is most often used as the front end (but could also be used separately as the back-end data store). The third is a component that is used in a three-tier or multi-tier application—Visual FoxPro is most often positioned as the "middle tier" between a user interface of indeterminate origin and a back-end data store. This middle tier contains the business logic and heavy-duty processing.

In this chapter, I'm going to describe what a LAN application looks like, discuss what each piece is and what tool we'll use to build it. Later in this book, I'll do the same for client-server apps and middle-tier components built with Visual FoxPro. But by the time you're done with this chapter, you'll understand how all of the various tools I've discussed fit in with each other and what part they play in building true Visual FoxPro applications.

All but the most trivial Visual FoxPro applications consist of multiple files. In the olden days, programmers would jam an entire application into a single program file. As this file grew larger and larger, and started to share code with other applications, this single program file referenced a common library of some sort or another.

Then the programmer realized that it was more efficient to have multiple program files that each were responsible for a specific function, so the application became a "main" or calling program, a common library, and multiple program modules.

These days, a Visual FoxPro application consists of one or more program files, perhaps a common library, but also menus, forms, and class libraries. True, you can convert forms and class libraries into program files, but even if you do, many more components in an application can't be converted to a program—such as reports, libraries, tables, and so on. Thus, even if you wanted to, you couldn't convert all of the components into a single program file.

A traditional FoxPro application consists of a "main" program that "sets the stage" for the application, runs a menu, and then goes into a "wait for event" state. The menu stays active, waiting for user input. Each option on the main menu either calls a submenu or a program, form, or report. These programs, forms, and reports may call additional programs, forms, or reports, and may reference other programs in a shared library of common routines. Think of these common routines as "ladies in waiting" or "butlers," available to provide services—of the programming kind—to the rest of the application. For example, do you need a number

formatted a certain way? A username and password validated? Data on a standard data-entry form to be inserted into a table? These common routines can be called on by the rest of the application to perform these types of tasks.

A Visual FoxPro application works in much the same way, except that instead of a common procedural library providing “services” to the rest of the application, a series of objects (yes, the type of objects one thinks of when discussing object-oriented programming) that have been created in the “setting the stage” part of application startup provide those services.

You very well may have looked at the sample Tastrade application that comes with Visual FoxPro. (If you can’t find it, you’ll need to do a bit of spelunking through the directories. Make sure you installed MSDN when you installed Visual FoxPro. Find the MSDN directory, likely under Program Files/Microsoft Visual Studio, and then drill down through 98VSA, 1033, Samples, and finally to VFP98. There you’ll find a series of directories for many different samples, including Tastrade.)

If you did, you probably followed these steps:

1. Run the application TASTRADE.APP, under the Tastrade directory, investigate the various menus, and play with the various forms for a while.
2. Investigate the Behind the Scenes forms in various places. (Under the Administration menu pad, there’s a Behind the Scenes menu option that allows you to look under the hood in many different places.)
3. Decide to look at the code yourself. Change to the Progs directory and look at the few program files. Then change to Forms and open the forms. Then switch over to Libs and open a few class libraries. Go back to the root and open the project.
4. After finding the top-level program (it’s MAIN.PRG under Programs in the Code tab), find code that looks like this:

```
--- Set up the path so we can instantiate the application object
IF SetPath()
  PUBLIC oApp
  oApp = CREATEOBJECT("Tastrade")
  IF TYPE('oApp') = "O"
    --- Release all public vars, since their values were
    --- picked up by the Environment class
    RELEASE gcOldDir, gcOldPath, gcOldClassLib
    RELEASE gcOldTalk, gcOldEscape
    oApp.Do()
  ENDIF
ENDIF

CLEAR DLLS
RELEASE ALL EXTENDED
CLEAR ALL
```

5. Shake your head, wondering where the application actually launches, since there doesn’t appear to be any menu, any foundation read (if you’re acquainted with them from FoxPro 2.x), or any other mechanism that “gets things going.” But you’re sure this is where things have to happen, because near the end of this code snippet you see things being released—the program must be finished by now.

6. Try to weed your way through the class libraries and forms again. Fill page after page with diagrams and maps of programs, forms, and class methods. Finally run out into the street screaming. If you're used to Xbase or other procedural languages, this approach—oApp = createobject("Tastrade") and oApp.Do()—turns the world on its ear.

It's quite a jump to get from Point A, a series of hierarchical programs that call one another off of a menu, to Point B, this set of objects floating around in the ether, hovering at the virtual shoulder of the application, willing to serve as needed. To describe this in one step would be much like discussing the use of a hammer, a saw, and a router, and then building a 10-story office building downtown. How about if I start out a bit slower, so you can see what is really happening when I appear to wave my hands? Good, I'd much prefer to do so as well.

## Defining the application

So, it's time to build an application. But what kind of an application am I going to build? What is it going to do, and what is it going to look like? You could write a book on the process of developing an application specification, and, indeed, I have—see the *1999 Software Developer's Guide* if you want the full story on specs. For the time being, however, I'm going to assume that the hard work of assembling a specification has been done, and I'll just follow the pictures.

I'm going to start by building a classic order-entry system, with customers, orders, detail lines, and invoices. Though simple, it will have enough pieces in it to demonstrate the different things I need to show you. Furthermore, just about all of you have seen one yourself, so you're not going to have to worry about learning the ins and outs of a new application—you'll be able to pay attention to just how to build it. And, finally, I'll eventually port this same application to a client-server framework, and then to a three-tier, Web-based structure as well. You'll be able to pay attention to the concepts each time.

## Deciding on a menu style

While most Windows applications you see these days use the same set of File, Edit, Window, and Help menu pads, this wasn't always true. There are several distinct menu styles floating around, and even now, in late 1999, I've run into menus that are throwbacks from the mid-1980s.

### Function-centric menu styles

A lot of custom and vertical-market database applications group functional areas and give each area its own menu pad on the main menu. Thus, you'll see an application with menu pads for Customer Service, Purchasing, Inventory, and Estimating, along with, perhaps, pads for System and Exit. Each pull-down menu would then have options that related to that functional area. Often, separate functions would be handled by different menu options and even different screens, so you'd see menu options for Add Customer, View Customer, Delete Customer, and so on.

This style is a legacy of the days when building menu bars was impossible with the tools available at the time. Applications at that point presented a list of choices in the center of the

screen. When menu building became a reality through a set of menu commands or through a visual tool like the Menu Builder, applications that used the old style of menu bars were ported over and the selections were just rearranged on the top of the screen. In other words, developers weren't taking advantage of new capabilities—they were just using new tools to do things the old way, much like the lumberjack used a chainsaw to manually saw through trees.

This style, however, may still be applicable in some situations. If you've got a large number of users across multiple departments and functional areas but they cross into each other's paths a lot and thus need access to a large number of menu options, this type of structure could be the best way to keep the myriad of choices organized and easily available to the user base. In fact, as I'm writing this book, I'm building an application that will be used by an entire company, and we're going with this structure for this very reason.

## **CUA menu styles**

“CUA” stands for Common User Access and is the standard for Windows applications. It dictates that the main menu will contain File, Edit, Window, and Help menu pads, each opening to show one or more menu options.

Forms that allow the user to maintain customers, orders, and invoices will be launched from menu options in the File menu. In this drop-down menu, options can be grouped according to function, with all customer-related forms in one section and all order-related forms in another. Or you might choose to group the options according to frequency of use, where the most commonly used forms are launched from a group near the top of the menu, and infrequently used options reside near the bottom, just above the Exit menu option.

Edit provides access to the standard Cut, Copy, Paste, Select All, and Find options.

Every good program provides a number of administrative and system-specific options. These might include the ability to change the configuration and settings of the application (System Preferences), data repair and recovery options, and user maintenance such as User Preferences, Change Password, and User Maintenance. These go under the Tools menu.

The Window menu pad simply contains options to manage a multiplicity of windows open on the desktop—Tile, Cascade, and Close All, for example—as well as a list of all currently open windows.

The Help menu pad provides system-supplied and user-defined help, system diagnostic information, and the traditional “About...” dialogs.

Of course, this isn't enough to build a complete application, but these are the standard menu pads. In a database application, you'll probably want at least two more.

The first, Processes, goes to the right of Edit, and is used to hold options that run operations that don't need much or any user intervention. For example, month-end posting routines, data imports, or matching processes would all be found under Processes.

The second, Reports, goes between Processes and Tools, and, obviously, is used for getting data out of the system. Thus, the word “reports” is actually a bit misleading, because you might not want only ad-hoc, custom, and hard-coded reports, but also any other type of output, such as exports to other applications, output to file formats that are picked up by other computers, or even just querying functions that will typically not be printed.

## Document-centric menu styles

The Tastrade application demonstrates a third style that has become somewhat popular over the past few years, although it hasn't exactly dominated application design. The view of the computer as a "document processor" has been increasingly espoused over the last few years, and in order to make the view all-encompassing, the definition of a document has been bent out of shape to accommodate all sorts of data structures that normally wouldn't have fit. It used to be that a document was essentially a single file, such as a spreadsheet, a memo, a book report, image, or a presentation. Each of these was a discrete file that could be found through the File, Open menu option, and then manipulated with Save, Close, Print, Send, and so on.

This doesn't work as well when you try to include data from database applications. What, exactly, does a "document" consist of? A record? A field? Multiple fields but not an entire record? Fields from multiple records? Multiple records from a single table? Each of these examples could easily be mapped to a printed document, so it's obvious that none of these is an all-inclusive definition. And what if your database application had to handle non-relational data, such as multimedia, e-mail, and other files? The concept of a "document" becomes more and more difficult to pin down.

The solution is not to define a document independent of the data stored on the computer, but instead along the lines of what a user sees. An invoice, then, could be a "document" even though it probably draws one or more records from up to a half-dozen tables. An e-mail message could be another document even though it's actually stored as sequential text in a BLOB (binary large object). The key is defining the document according to data usage.

Thus, a true document-centric menu system looks a lot like that used by applications such as Word, Excel, and PowerPoint. You'll open a form with function-specific menu options, like those found under the Customer Service, Orders, and Physical Inventory menu pads of an older system, but then you'll use menu options under the File menu pad to manipulate the form and its data. Clicking New will create a new "document" (an invoice, a customer, a shipping manifest), Save and Restore will either commit modifications to the rest or revert to the old data, and Delete will delete the document. Close will close the form, and Print will bring up a print dialog box that provides a number of document-specific choices. The toolbar provides an alternative mechanism to execute many of these functions. Navigation through a table is provided by the functions in the toolbar as well as under a Navigation menu pad (which I think is a little contrived).

A twist on this paradigm is to include an Open menu option under the File menu pad as well, and then display a dialog where users can choose what they want to open: an invoice, customer, shipping manifest, inventory record, and so on, much like in Word you can open an ordinary document, a template, or other document types.

At first I hated this style, but that's often the case with something new, especially when it flies in the face of what you've personally constructed. The more I worked with it, however, the more I began to like certain elements. Still, the interface isn't quite consistent and could be improved. As I mentioned, the navigation seems awkward; it's very clearly a record-navigation tool, but I thought we were using documents.

Nonetheless, this document-centricity is certainly worth looking at—some of you might jump up and down, thinking that at last you've found the Holy Grail to interfaces, while others of you will decide to keep looking.

The key point to remember about selecting and designing a menu, as well as the rest of your user interface, is that your users will learn it faster and make fewer mistakes if it works like something else they know. Remember when you had to remember a dozen different ways to save your work, depending on whether you were using one program or another? Now, Ctrl+S pretty much saves every type of document you could be working on in Windows—you can use your brainpower to remember important things now. If you use a paradigm that is familiar to your user base, they will come up the learning curve easier than if you introduce something radically different. There has to be a significant advantage to a new interface to make it worth your users' while.

### **A pseudo-object-oriented menu style**

One last issue that you might want to think about when determining how you're going to build your menus is reusability. As the trade press has been telling us for years, the real revolution in software will come when object orientation is a reality. Being able to create an object once and then reuse it will greatly enhance your productivity in a number of ways: eliminating repetitive coding, reducing errors by using tried and tested code, eliminating testing of parts that have already been approved, and keeping your adrenaline level high because you're always working on new and existing projects instead of slogging through near-mindless grunt work.

Consider a menu structure where there are absolutely no custom menu options. Instead of listing the various forms under one or more menu pads, place them in an array—perhaps according to availability as determined by the permission level of the user—and then access that array with the File, Open menu option. When users select File, Open, they are presented with a list of available forms, as I described earlier. In other words, you have a menu object that operates on data that is passed to it.

It doesn't matter how the internals of the menu work. You can give this menu object to another developer, along with the explanation of how to pass the names of the forms, and they can build applications without ever seeing the inside of the menu. You can do the same thing yourself—use it in other applications of your own just by passing it different sets of data. No more boring coding of menus!

I'll take this one step further. It is conceivable that an application could have dozens and dozens of forms. At some point, you'll probably need a whole new interface to handle these gazillions of forms. However, there's a middle ground where you don't want to go to that effort, but still can't just list 34 forms in a long list—it would end up looking like the Programs menu list from the Windows Start button. An alternative would be to break the forms into functional groups and then list the appropriate forms under each group, much like a directory/subdirectory hierarchy.

It wouldn't, then, be a terrible reach to change the File, Open mechanism to read either a single list of names or a two-column array that would contain the name of the group in one column and the name of the form in the second column. Selecting the File, Open menu option would list all of the groups, and upon the selection of a specified group, all of the available forms attached to that group would be displayed. It would operate identically to the section of a document in a directory hierarchy, so it's a very comfortable, common interface for the user (well, at least for most users). And this takes our object-oriented analogy one step further. We're now using polymorphism—using the same “verb” to do two different operations, depending on the context in which it's used. If we're in the context of a single-column array,

File, Open just means “open the selected form”; if we’re in the context of a two-column array, File, Open means “display a list of groups first, and then display the forms for a selected group.”

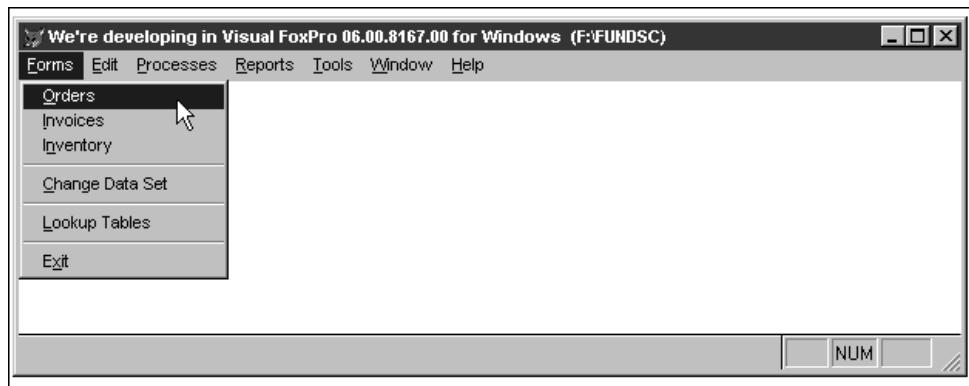
But that’s not all, of course. If you decided to go with a document-centric menu style, the same File, Open paradigm would still work. And you could use a different relationship than groups and forms—how about displaying just those forms (or documents) according to which security level the user belonged to? The possibilities are nearly endless.

## The Books and Software Inventory Control Application menu structure

As with all parts of this application, before I can write it, I have to describe it. Here’s the menu specification.

### Forms menu

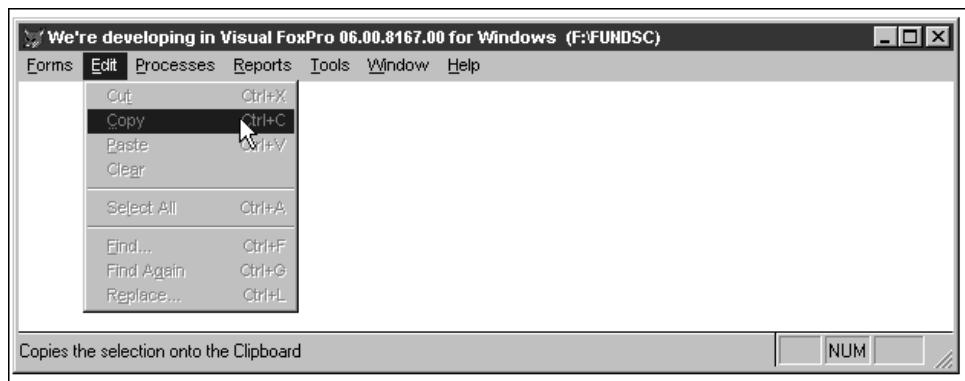
The Forms menu includes an option for each form in the system, a menu option that calls a generic lookup maintenance form, a way to switch from one data set to another, and a way to exit the application. By default, each form can be opened multiple times, and each time it is opened, its name (followed by a number if it’s already open) will be placed on the Window menu. See **Figure 12.1**.



**Figure 12.1.** The Forms drop-down menu.

### Edit menu

The Edit menu contains the standard Cut, Copy, Paste, Select All, and Find options. The primary reason this menu pad exists is to provide this functionality through shortcut keys within text and edit boxes on forms. You must have the menu defined in order to make the corresponding keyboard shortcuts available. See **Figure 12.2**.



**Figure 12.2.** The Edit drop-down menu.

## Processes menu

The Processes menu contains options for a couple of processes that are run on a regular basis. One process is importing e-mail from various customers and vendors who send electronic notification of what they've sold or what they want to order. Instead of retying this information, the application will automatically import that information as needed.

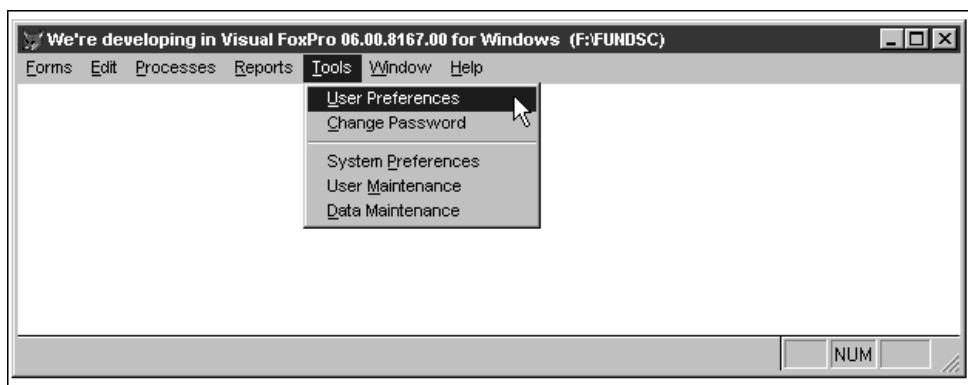
## Reports menu

What good is an application without reports? A few reports are included in this system, as would be expected in any good application.

## Tools menu

The Tools menu provides access to utilities and tools for both the user and the system administrator. The user utilities include User Preferences, where users can configure the application to their own preferences, such as setting and changing their passwords; and System Info, where the user can display a series of status screens that show various information about the configuration of the machine, software, and application. This option is provided to the user for diagnostic purposes.

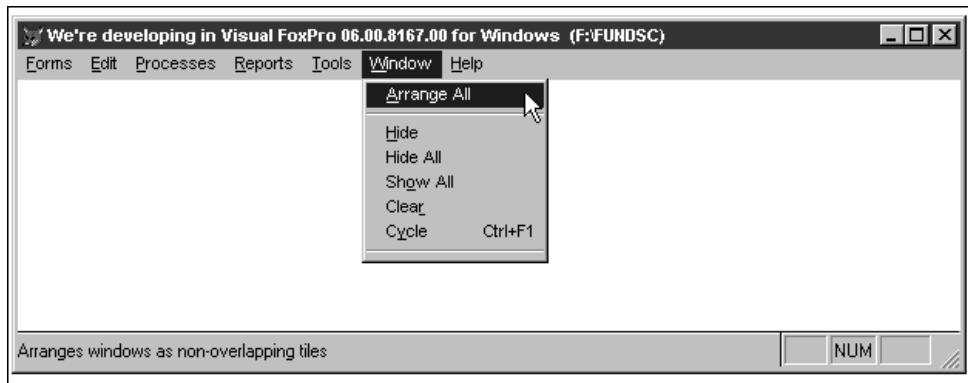
The system administrator menu options include System Preferences, which allows the user (usually an administrator) to specify default options for new users and a baseline to which other users can revert their settings. User Maintenance allows the system administrator to add, edit, delete, and lock out users. Data Maintenance allows the user to perform a number of functions relating to the integrity and validity of the data, including reindexing, packing memo fields, removing deleted records from the tables, and validating the primary and foreign keys in the database. See **Figure 12.3**.



**Figure 12.3.** The Tools drop-down menu.

## Window menu

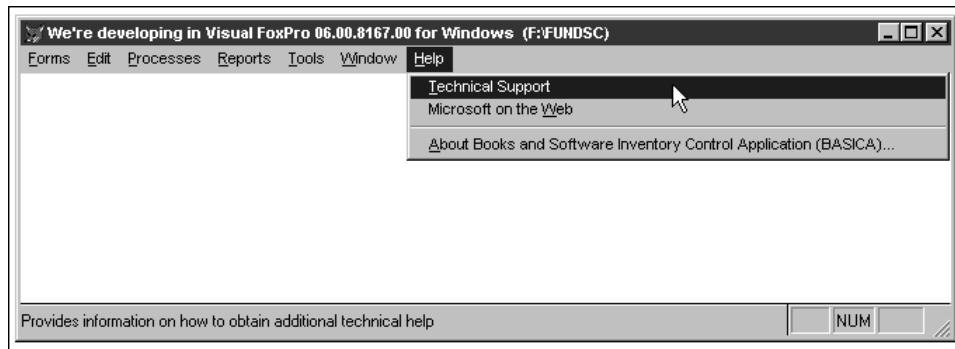
The Window menu option contains a list of the open windows, as well as functions to operate on those windows during the execution of the application. If no forms are open, the Window menu pad is disabled (although I've enabled it in **Figure 12.4**).



**Figure 12.4.** The Window drop-down menu.

## Help menu

The Help menu contains Help-related options. These include a list of Help Topics, User-Defined Help (if you choose to provide it), and the omnipresent "About..." dialog boxes that contain copyright notices and generic information about the application such as the version number and registered user. See **Figure 12.5**.



**Figure 12.5.** The Help drop-down menu.

Before you get too excited, you need to know that I'm not actually going to build every one of these pieces in the application. Rather, I'll stub out many of these options with placeholders. I've gone through each of these descriptions to give you an idea of some of the standard functionality you might want to include in each of your applications. I have built a menu with hooks for each of those functions already built in. I'll build the menu and a few simple forms in the next couple chapters, but that's all. The rest of the work, as they say, is left as an exercise to the reader.



If you're following along on your computer and are ready to start creating your own menu, I might suggest that you create a dummy menu by using the Menu, Quick Menu option, and then modify the result as needed, instead of starting with a menu from scratch. If you want a menu that's already been baked and packaged, check out the project, named IT, and the menu, also named IT, in the Chapter 12A source code.

As an aside, I name the project, main menu, and top-level program of every application of mine "IT", so that the source code directory of each application contains the following files:

```
IT.PJX
IT.PJT
IT.PRG
IT.MNX
IT.MNT
```

There are a number of different naming schemes around—and I don't like any of them. Some developers call their top-level program "MAIN," while others use a custom name based on the name or the acronym of the system. I don't like the first choice because it's easy to get confused about the calling program. I've run into plenty of systems that had the following files in the source code directory:

```
MAIN.PRG
MENU.PRG
MAINMENU.PRG
MENU1.PRG
MAIN1.PRG
MENUTOP.PRG
MENUMAIN.PRG
```

---

Sure, you can determine which program is the top-level routine by looking in the project—that is, if the project files are still around. But I've further found the following project files in the same source code directory:

```
MAIN.PJX
MAIN.PJT
MENU.PJX
MENU.PJT
MAINMENU.PJX
MAINMENU.PJT
MENU1.PJX
MENU1.PJT
MAIN1.PJX
MAIN1.PJT
MENUTOP.PJX
MENUTOP.PJT
MENUMAIN.PJX
MENUMAIN.PJT
```

Now what do you do? You have to examine the date/time stamps (hoping they're all correct), or open the files themselves.

The alternative is to use the acronym of the system as the name of the project and top-level program. I always forget the acronyms, and I sure don't like doing a lot of typing. Simply using "IT" clears away a lot of these issues, because no matter which application you are working on or running, you can enter:

```
modify project it
```

or

```
do it
```

and everyone is happy.

## Running a menu

Now that you've got a menu (whether you built the menu yourself or just copied the one provided with the source code), you might be tempted to run it—and indeed you can. First you must generate it, and it's easiest to do that by opening the Menu Builder and then choosing the Menu, Generate command. The Menu Generator creates a program file, called IT.MPR, from the menu files (IT.MNX and IT.MNT). When you execute IT.MPR, Visual FoxPro will compile the program, create an object file called IT.MPX, and execute that. Thus, to run your menu once you've built and generated it, just issue the command:

```
do IT.MPR
```

You'll see the Visual FoxPro menu replaced by your menu. Unfortunately, you'll also see the Command window return, along with any other windows that you had open at the time. Even worse, you can't get the VFP menu back—what do you do? Issue the command:

```
set sysmenu to default
```

Well, that solved the problem of the disappearing VFP menu, but you're still stuck with a fairly useless menu.

## **Building a main program and an Event handler to hold up the menu**

Maybe the problem is that you have to write a program to run the menu! That could be it! Well, actually, it isn't. If you want, you could build a program called, say, ITBAD.PRG, with the following code:

```
* itbad.prg
* clean up a bit before we start
close all
clear all
* run the menu
do IT.MPR
```

If you've tried this, you've found that it doesn't work any better. The menu replaced the Visual FoxPro menu, but the Command window came back, and you're in the same place you were a minute ago.

Here's what's happening.

Think about a macro that you've assigned to a keyboard hotkey. You can think of the macro as a program, but it's not an executing program. It's merely sitting in memory, waiting for you to execute it. How do you execute it? By calling it from a keyboard hotkey or shortcut, like Ctrl+F12 or Alt+S. How did those keyboard shortcuts get set up? You ran some sort of program, possibly even from within the macro builder program, that set up the shortcuts. But the crux of the matter is that the macros aren't running—they're sitting around waiting for you to call them. This situation is called a "wait state" and your computer acts as an "interrupt handler"—it goes and does its thing until, well, until when? Until you press one of the keyboard shortcuts, interrupting the computer from whatever it's doing, and making it run the macro (program) assigned to your keyboard shortcut.

A Visual FoxPro menu is pretty much the same thing. When you ran your menu program, you in essence set up a series of hotkeys—available not only to the keyboard but also to the mouse—and that's it. Executing those hotkeys or mouse clicks will run the programs attached to them, just as executing the macro hotkey described in the previous paragraph will run the macro.

You basically painted a picture at the top of your computer screen and told Visual FoxPro to wait for certain mouse clicks or keyboard shortcuts. Once the picture was painted, control was returned to your regularly scheduled program—that is, Visual FoxPro—via the Command window. Only thing is, now you have a menu with different functionality. Sure, you can run the various forms, reports, and other programs from this new menu, but you can also do anything else you want, via the Command window—and so can your users!

What is needed here is a mechanism to "hold up the menu"—that is, to trap for the events that the menu will process, but keep away from Visual FoxPro's interactive environment, until

the user chooses to quit the application. In other words, a “wait state” for menu commands has to be introduced to Visual FoxPro on top of the natural wait state provided by Windows.

In previous versions of FoxPro, the developer had to go through a number of gyrations to accomplish this. With Visual FoxPro, the simple READ EVENTS command is all that is needed.

Including this command after the command that executes the menu program will cause Visual FoxPro to process all of the keyboard and mouse events a user can produce. This is called an “event handler” and that’s what it does—it handles all of the possible (computer-related) events until it’s told not to anymore.

The sister command CLEAR EVENTS is attached to the Exit menu option; it tells Visual FoxPro to terminate the event handler—or, in other words, to destroy the wait state and continue processing the program with the line of code that follows the READ EVENTS command.

## Enhancing the main program

So now our top-level program, IT.PRG, looks something like this:

```
* IT.PRG
close all
clear all
do IT.MPR
read events
close all
clear all
```

Though necessary, this isn’t really a sufficient program. Here’s the functionality that a top-level program really ought to contain:

1. Save the current Visual FoxPro environment.
2. Read in the application settings.
3. Check the hardware and operating system environment.
4. Set up the application’s environment.
5. Handle user login and set up security.
6. Set up the user interface and run the main menu.
7. Set up the event handler.
8. Upon cancellation of the event handler, set everything back to the way it was.
9. Shut down the application (and Visual FoxPro).

You could write a bunch of procedural code to do all this, and throw it in the top-level program. If you did, here’s what it might look like:

```
* it.prg
* top level program
```

```
*  
* set stuff up  
  
* talk is a special case  
if set("talk") == "ON"  
  set talk off  
  m.gcOldTalk = "ON"  
else  
  m.gcOldTalk = "OFF"  
endif  
* save current environment  
m.gcOldCent = set("century")  
m.gcOldDelete = set("deleted")  
m.gcOldEscape = set("escape")  
m.gcOldExact = set("exact")  
m.gcOldExclusive = set("exclusive")  
m.gcOldMult = set("multilocks")  
m.gcOldProc = set("procedure")  
m.gcOldRepr = set("reprocess")  
m.gcOldSafe = set("safety")  
m.gcOldStat = set("status bar")  
m.gcOldHelp = set("help",1)  
m.gcOldReso = sys(2005)  
m.gcOldOnEr = on("error")  
* clean out everything  
release all except g*  
close all  
clear menu  
clear popup  
clear window  
clear  
* set up outside resources  
* (stubbed out for this example)  
*\\set classlib to ("MYBASECLASSES")  
*\\set help to ("MYHELP")  
set procedure to ("MYPROC")  
*\\set resource to ("MYRES")  
* initialize various system attributes  
* by reading in from outside the application  
* (either an INI file or the Windows Registry)  
* (GetRegValue is a function in MYPROC)  
m.gCanGoOn = .t.  
m.gcNameSystem = GetRegValue("NameSystem")  
m.gcVersion = GetRegValue("VersionSystem")  
m.gcDefaultDataLocation = GetRegValue("DefaultDataLocation")  
m.gnMinRAM = GetRegValue("MinRAM")  
m.gnMinDiskSpace = GetRegValue("MinDiskSpace")  
* make sure the default data set exists and is good  
if !DataIsGood(m.gcDefaultDataLocation)  
  m.gCanGoOn = .f.  
endif  
* check the hardware and OS  
* (each of these functions is in MYPROC)  
m.gnAvailRAM = GetRAM()  
  
if m.gnAvailRAM < m.gnMinRAM  
  m.gCanGoOn = .f.  
endif  
m.gnAvailDiskSpace = GetDiskSpace()
```

```
if m.gnAvailDiskSpace < m.gnMinDiskSpace
  m.glCanGoOn = .f.
endif
* initialize user login parms
m.gcNameUser = "ADMIN"
m.gcPassword = "ADMIN"
m.gcPermLevel = "00001"
m.glLoginWasGood = .t.
* get user login
do form LOGIN
if !m.glLoginWasGood
  m.glcCanGoOn = .f.
endif
* set up On Error and On Escape values depending
* on if user has developer-level permissions
if m.gcPermLevel = "ADMIN"
  on error do DevError
  set escape on
else
  on error do UserError
  set escape off
  on escape *
endif
* define environment
set century on
set deleted on
set exact off
set exclusive off
set multilocks on
set reprocess to 5
set safety off
*
* set up event handler
*
* save old menu, run new one
if m.glcCanGoOn
  *** push menu _MSYSMENU
  do IT.MPR
  read events
else
  * something bad happened so can't run app
  * (probably want to let the user know what)
endif
*
* clean up and close shop
*
* upon termination, restore menu
***pop menu _MSYSMENU to master
set sysmenu to default
* return environment
set century &gcOldCent
set deleted &gcOldDelete
set escape &gcOldEsca
set exact &gcOldExac
set exclusive &gcOldExcl
set multilocks &gcOldMult
set procedure to &gcOldProc
set reprocess to (m.gcOldRepr)
set safety &gcOldSafe
```

```
set status bar &gcOldStat
if file(m.gcOldHelp)
  m.cTemp = "" + gcOldHelp + ""
  set help to &cTemp
  rele m.cTemp
endif
if file(m.gcOldReso)
  set resource to &gcOldReso
endif
on error &gcOldOnEr
* clean out everything else
close all
clear menu
clear popup
clear program
clear window
clear
set talk &gcOldTalk
* from within VFP or from runtime
if version(2) = 0
  wait window nowait "See ya later, alligator"
  clear memory
  quit
else
  @2,0 say "See ya later, alligator"
  clear memory
  return
endif

* <EOF>
```

And here are the dummy calls in MYPROC:

```
* MYPROC.PRG

function GetRegValue
lpara m.tcDummy
* placeholder for sample function in IT.PRG
return .t.

function GetRAM
lpara m.tcDummy
* placeholder for sample function in IT.PRG
return .t.

function GetDiskSpace
lpara m.tcDummy
* placeholder for sample function in IT.PRG
return .t.

function DataIsGood
lpara m.tcDummy
* placeholder for sample function in IT.PRG
return .t.

procedure DevError
* placeholder for sample proc call in IT.PRG
return .t.
```

```
procedure UserError  
* placeholder for sample proc call in IT.PRG  
return .t.
```

Most of this code is pretty straightforward, but a couple of things bear mentioning. First of all, when I'm done testing an application, I want to return to the Visual FoxPro environment from which I came. In other words, if I started in the interactive environment, I want to return there, but if I started from a Windows executable, I want to return to Windows. In the code for the exit, I check the VERSION() function, and either return to VFP or quit.

Another thing is that I typically modify a few menu options depending on which user is logged onto the system. I'll include a Developer menu pad that is active when a user with a developer permission level logs in. (Actually, it's always there, and I remove it if a non-developer logs in.) I'll also remove the system administrator menu options in the Tools menu if a non-administrator logs in. Here's how.

Suppose you've decided on a permission-level structure where a developer has a permission level of "00000", an administrator has a permission level of "00001", and so on. Open the Menu Builder, and then open the General Options dialog by selecting the View, General Options menu option. Select the Cleanup check box and click OK to open the editing window. Enter the following code into the editing window:

```
If !(m.gcPermLevel == "00000")  
    Release pad Developer of _MSYSMENU  
Endif  
  
If !inlist( m.gcPermLevel, "00000", "00001")  
    Release bar 3 of Tools  
    Release bar 4 of Tools  
    Release bar 5 of Tools  
    Release bar 6 of Tools  
Endif
```

Now save the cleanup code and regenerate your menu. This code will be included at the end of the menu program so that when the menu has finished running, the last thing the menu program does is remove those pads and bars that were previously defined. If you run this program on a really, really slow computer, you can actually see the entire menu being built, and then the Developer menu pad being removed.

The next-to-last step is to put this top-level program and the menu into a project. As you might remember from Chapter 6 on the Project Manager, once you identify the top-level program, you can execute the Build option and have the Project Manager automatically find all of the files. Thus, create a project called IT, add the top-level program, IT.PRG, to the Programs topic in the Code tab, and click the Build button. The IT menu will automatically be included (and regenerated as well).

The last step, then, is to run the program. Because the top-level program is called IT, you can simply execute the command:

```
do it
```

And your application will run.

Notice that there isn't any application-specific code in this program, and precious little in the menu either. If you decide to use a File, Open paradigm instead of hard-coding menu options, you might never touch your top-level program or your menu again!

## Starting the move to OOP

While this procedural method for application startup works, there's a better way to do this. You can build a series of objects that handle all of these responsibilities—the environment, security, the event handler—and then call these objects. In the rest of this chapter, I'll show you how to move from this huge top-level program to an elegant calling routine that simply sets up the objects and lets them do all the work.

If you recall, at the beginning of the chapter I showed you a chunk of code like this:

```
--- Set up the path so we can instantiate the application object
IF SetPath()
  PUBLIC oApp
  oApp = CREATEOBJECT("Tastrade")
  IF TYPE('oApp') = "O"
    --- Release all public vars, since their values were
    --- picked up by the Environment class
    RELEASE gcOldDir, gcOldPath, gcOldClassLib
    RELEASE gcOldTalk, gcOldEscape
    oApp.Do()
  ENDIF
ENDIF
```

So far, I've done nothing that even remotely looks like this. Sure, you've seen the word "environment" a few times, but that's not really close. So how does this procedural menu code relate to this object-oriented style? Well, depending on your frame of mind, it's either a huge jump or just a short hop.

The application framework has a number of generic functions, right? I'm going to turn these functions into classes that belong to a generic application class library. The syntax will change, and the extensibility will expand greatly, but the conceptual jump is fairly straightforward. Each generic function will correspond to a class or a method in the application class library.

Take the issue of saving the environment settings—it's always polite to return what you've borrowed in the same condition. And because the application could be considered to be "borrowing" the Visual FoxPro environment while it's running, it would be nice to return the environment from whence it came. And you could do this in one of two ways.

You see, there are well over 100 SET commands in Visual FoxPro—close to 200 if you include all of the permutations and parameters that come with the SET function. Many of these are used for backward-compatibility or in extremely unusual circumstances. So while you might save the current status of every one of them, and then set each one to the way you want it, that's going to take a few seconds during application startup, and that's a few seconds too many. Instead, being pragmatic, most developers have a "Top 10" list of SET functions that they use. The trouble is, this list gradually expands over time as you run into new situations or as the language grows.

This is a typical example of having 50 versions of the SAVE\_SET procedure at various installations and eventually running into maintenance hell. Thus, it's a classic place where you might use this object-oriented stuff instead. Because there's nothing to "see" when you save the current settings and later restore them, it's also a classic "non-visual class" example. I'll show you how to build this class—and how to implement it in the application.

## A quick review of non-visual classes

First, remember that you can think of a non-visual class as simply a set of procedures and initialized variables. Next, remember that a class library is a table—with an extension of .VCX—that contains one or more records. And because a class library contains one or more classes, a class is made up of one or more of these records in the .VCX. Visual classes generally have multiple objects, and each of these objects can have many properties and methods—and so each object requires its own record. However, unlike visual classes, non-visual classes are rather compact. All of the properties and methods in a non-visual class are contained in the same record. Therefore, a non-visual class consists, in general, of a single record in a .VCX. And thus, a non-visual class library can contain multiple non-visual classes.

## Creating a non-visual class

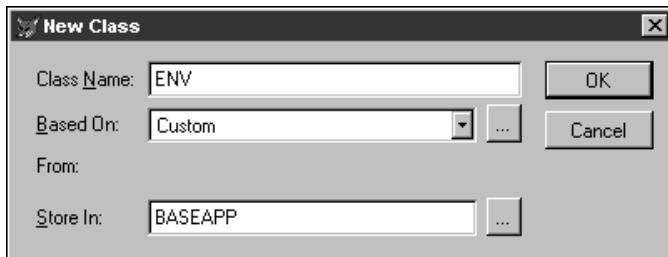
Okay, that's a lot of words and theory—how about a real-life example? I'm going to create a class library called BASEAPP.VCX, and it will contain several non-visual classes. The class I'm concerned with now will handle our environment saving and restoring, and I'll call it ENV. (Remember, though, that the class library BASEAPP.VCX will also contain other classes, which I'll build later in this chapter.) This class will contain a number of methods (they're just subroutines, right?) that perform the actions of saving the original SETs, changing the SET variables to the values we need for our app, and then, when the app is finished, changing them back.

These methods will be called SaveSets(), DoSets(), and RestoreSets(). Furthermore, I'll need a "place" to store the SETs settings that are captured via SaveSets(). These will be properties of ENV (they're just variables, right?) named "cOldCent", "cOldSafe", and so on. If you look at IT.PRG above, those look familiar, don't they? Thus, the ENV non-visual class will consist of a single record in BASEAPP.VCX, and that record will have fields that hold the various methods and properties of ENV.

 Here's how to create the class library BASEAPP.VCX and then create ENV. You can find the source code to these steps in the Chapter 12B files for this book.

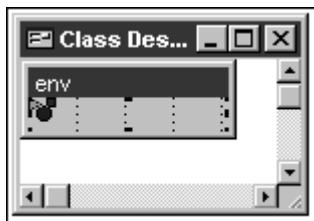
1. Create a new class by issuing the CREATE CLASS command, or by selecting the File, New, Class menu option, or by selecting the Classes node in the Project Manager and clicking the New button. The New Class dialog appears as shown in **Figure 12.6**.
2. Enter the following:
  - Class Name: ENV
  - Based On: hwcustom
  - Store In: BASEAPP.VCX

Note that if BASEAPP.VCX does not exist, it will be created for you.



**Figure 12.6.** Use the New Class dialog box to create a non-visual class called "ENV" and store it in BASEAPP.VCX.

3. The Class Designer will appear and the object in the Class Designer window will be a funny-looking little gizmo with three shapes of various colors as shown in **Figure 12.7**. This indicates that it's a non-visual class, which just means we're not going to drag controls like command buttons and page frames onto the class.



**Figure 12.7.** The Class Designer displays a tri-colored shape when you are creating a non-visual class.

4. Open the Properties window:
  - Select the View, Properties menu option or select the Properties menu option from the shortcut menu that appears when you right-click on the object in the Class Designer window.
  - Select the Methods tab, and notice that there are only the default methods for a custom class.
5. Add the SaveSets() method to the ENV class:
  - Select the Class, New Method menu option.
  - Give it a name of "SaveSets."
  - Add a description like "Saves existing SETs" as shown in **Figure 12.8**.



**Figure 12.8.** Use the New Method dialog box to create a custom method to save settings that existed when the app was started.

- Click Add.
  - Click Close. (In previous versions of Visual FoxPro, you had to choose the Class, New Method menu option each time you wanted to add a property or method, which was quite a nuisance; now the dialogs stay open until you're done.)
6. Create the code for the method by opening the Code window as shown in **Figure 12.9**:
- Select the Code menu option from the shortcut menu that appears when you right-click on the object in the Class Designer window, or select the Methods tab in the Properties window and double-click on the SaveSets() method.
  - Enter the following code:

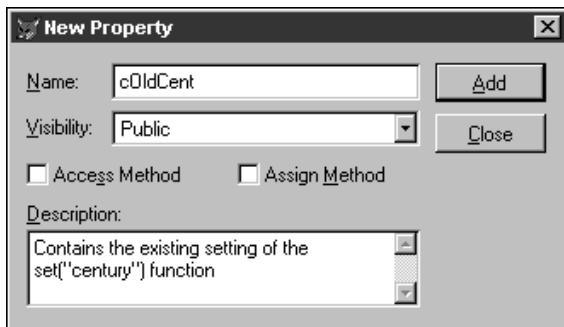
```
this.cOldCent = set("century")
this.cOldDelete = set("delete")
this.cOldEscape = set("escape")
this.cOldExact = set("exact")
this.cOldExclusive = set("exclusive")
this.cOldMult = set("multilocks")
this.cOldProc = set("procedure")
this.cOldReprocess = set("reprocess")
this.cOldSafe = set("safety")
this.cOldStatus = set("status bar")
this.cOldTalk = gcOldTalk
this.cOldHelp = set("help",1)
this.cOldReso = sys(2005)
this.cOldOnError = on("error")
```



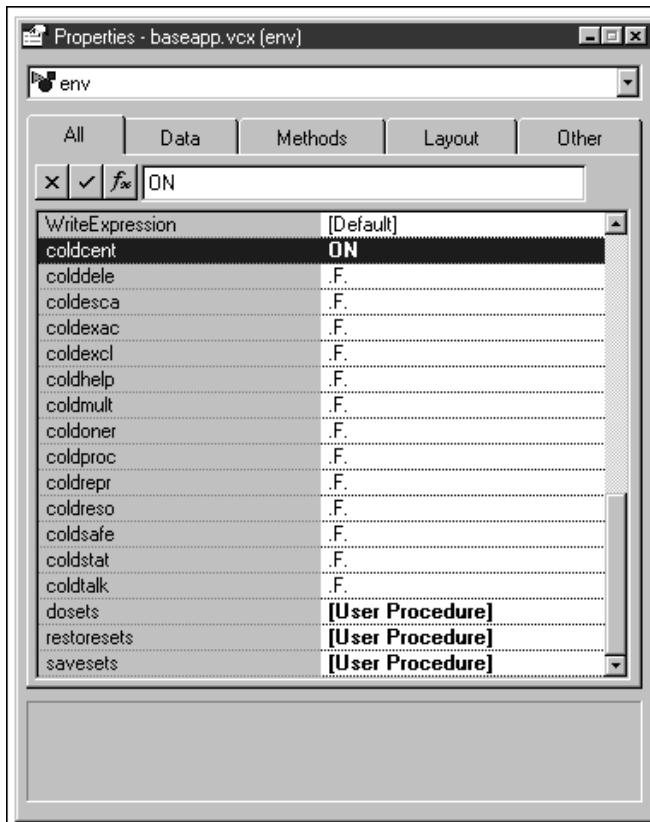
**Figure 12.9.** Enter the code for the method using the Code window.

Now you've got a method that grabs the current settings and stores them to—wait, where does it store them? What are these “this.cOldCent” and “this.cOldClas” variables? Well, actually, they're not “variables”; they're properties of this class. Remember that a property is just a variable that is stored along with the methods of the class. So just as you can create a variable inside a subroutine, you can create a property that's part of a class. And we're going to create our own properties—named “cOldCent” and “cOldClas”—to hold the values of the various settings. The “this” prefix in front of each property just tells the method that you're assigning values to the properties with these names that belong to this class, not some other class that also has a “cOldCent” property. So the next step is to create these properties.

7. Create properties for each of the settings we're going to store:
  - Select the Class, New Property menu option to open the New Property window as shown in **Figure 12.10**.
  - Give it a name of cOldCent.
  - Add a description like “Contains the existing setting of the set(“century”) function”.
  - Click Add.
  - Repeat this process for all the other properties. When you've finished the last one, click Add, then Close.
  - Initialize each property's value to be *character* as shown in **Figure 12.11**.



**Figure 12.10.** Use the New Property dialog box to create a custom property that will be used to save the value of a single setting.



**Figure 12.11.** Be sure to change the initial value of all new properties or set them to blank in the Properties window.

8. Add the DoSets() and RestoreSets() methods in the same way you added SaveSets().

These methods use the existing properties but don't require new properties. The code for the DoSets() method is:

```
set century on
set escape on
set clock status
set deleted on
set exact off
set exclusive off
set multilocks on
set reprocess to 5
set safety off
```

The code for the RestoreSets() method is:

```
* use "u" because you may be storing either
* character or numeric values to luTemp
* in your own routine
local luTemp

luTemp = this.cOldCent
set century &luTemp

luTemp = this.cOldClas
set classlib to &luTemp

luTemp = this.cOldDele
set deleted &luTemp

luTemp = this.cOldEsca
set escape &luTemp

luTemp = this.cOldExac
set exact &luTemp

luTemp = this.cOldExcl
set exclusive &luTemp

luTemp = this.cOldMult
set multilocks &luTemp

luTemp = this.cOldProc
set procedure to &luTemp

luTemp = this.cOldRepr
set reprocess to (luTemp)

luTemp = this.cOldSafe
set safety &luTemp

luTemp = this.cOldStat
set status bar &luTemp
```

```
luTemp = this.cOldHelp
if !empty( luTemp )
  set help to &luTemp
endif

luTemp = this.cOldReso
if !empty( luTemp )
  set resource to &luTemp
endif

luTemp = this.cOldOnEr
on error (luTemp)

luTemp = this.cOldTalk
set talk &luTemp
```

This seems like a lot of work, but this is the only time you'll ever have to do it.

## Implementing a non-visual class

Now that you've got your ENV class, how do you meld it with the application? First, you need to rip out the code in IT.PRG that used to do this, and then you need to have Visual FoxPro use this class to do the functions instead. Ripping the code out of IT.PRG isn't technically or conceptually tough, of course—it's the other part. You're going to perform a function that you might think of as opening a procedure or subroutine library in order to access the ENV class, but the syntax will be a little different.

Now that you've got a class called ENV, you can create an object from this class and then refer to the object and its properties and methods. Here's the code (and you can run this right from the Command window, after you save and close the BASEAPP class library):

```
set classlib to BASEAPP
oEnv = createobject("ENV")
```

The first command tells Visual FoxPro that you're interested in accessing the class library named BASEAPP. The second command looks inside BASEAPP and creates an object called oEnv. You can think of this as a memory variable—and in fact, you could put “oEnv” in the Debug Watch window and see what shows up when you run the commands. Now that you've got this object instantiated (in other words, you've created an instance of the class and named that instance “oEnv”, just as when you create a form from a visual class), you can access its properties and methods.

To determine the value of one of its properties, you could enter this in the left side of the Debug Watch window (without the question mark), but it's easier to simply enter it in the Command Window:

```
? oEnv.cOldCent
```

You'd see either .F. or ON as the result. Now this is sort of curious, isn't it? Why? Because the value of set(“century”) is either ON or OFF, not .T. or .F., right?

All you're doing now is evaluating the value of the property, and if you didn't initialize the property as described at the end of step 8 earlier, its value will be its default—which is always a

logical false. Thus, if you were going to compare oEnv.cOldCent to ON, you'd get a Data Type Mismatch error. Of course, you'll want to consider carefully how you initialize your properties. If you use a default value that might not be accurate, you could cause worse problems than a Data Type Mismatch error.

Similarly, to execute one of the objects' methods, call it like any other function:

```
oEnv.SaveSets()
```

Don't be worried if the method chokes on the line where it attempts to save the setting of a global memory variable, gcOldTalk. This memory variable wasn't created in this interactive session, but it will be in your program. Just ignore the error message and continue.

After you execute the SaveSets() method, each of the cOldXXXX properties of oEnv have been set to their current values in your Visual FoxPro environment. If you want to examine them, you can simply enter the following code in the Command window like before:

```
? oEnv.cOldMult
```

You could even change the values of those properties interactively, like so:

```
oEnv.cOldMult = "3.14159"
```

This is, of course, nonsense; this has just entered a ridiculous value in the cOldMult property. I just want you to get used to accessing an object's properties and methods.

Now that you've got this environment class and know how to create an ENV object, it's time to replace some of that procedural code in IT.PRG with calls to the various methods of oEnv—after, of course, you instantiate oEnv. Here's what IT.PRG looks like at this stage. I'm not done with this example, but at this point you've turned the corner from procedural programming to implementing OOP in your applications:

```
* it.prg
* top level program

*
* set stuff up
*
* talk is a special case
if set("talk") == "ON"
  set talk off
  m.gcOldTalk = "ON"
else
  m.gcOldTalk = "OFF"
endif

* clean out everything
release all except g*
close all
clear menu
clear popup
clear window
clear
```

```
* set up outside resources
set classlib to BASEAPP.VCX
oEnv = createobject("ENV")
* (stubbbed out for this example)
*\\ set help to ("MYHELP")
set procedure to ("MYPROC")
*\\ set resource to ("MYRES")

* save current environment
oEnv.SaveSets()
* define environment
oEnv.DoSets()

* initialize various system attributes
* by reading in from outside the application
* (either an .INI file or the Windows Registry)
* (GetRegValue is a function in MYPROC)
m.g1CanGoOn = .t.
m.gcNameSystem = GetRegValue("NameSystem")
m.gcVersion = GetRegValue("VersionSystem")
m.gcDefaultDataLocation = GetRegValue("DefautlDataLocation")
m.gnMinRAM = GetRegValue("MinRAM")
m.gnMinDiskSpace = GetRegValue("MinDiskSpace")
* make sure the default data set exists and is good
if !DataIsGood(m.gcDefaultDataLocation)
  m.g1CanGoOn = .f.
endif
* check the hardware and OS
* (each of these functions is in MYPROC)
m.gnAvailRAM = GetRAM()
if m.gnAvailRAM < m.gnMinRAM
  m.g1CanGoOn = .f.
endif
m.gnAvailDiskSpace = GetDiskSpace()
if m.gnAvailDiskSpace < m.gnMinDiskSpace
  m.g1CanGoOn = .f.
endif
* initialize user login parms
m.gcNameUser = "ADMIN"
m.gcPassword = "ADMIN"
m.gcPermLevel = "00001"
m.g1LoginWasGood = .t.
* get user login
do form LOGIN
if !m.g1LoginWasGood
  m.g1CanGoOn = .f.
endif
*
* set up event handler
*
* save old menu, run new one
if m.g1CanGoOn
  *** push menu _MSYSMENU
  do IT.MPR
  read events
else
  * something bad happened so can't run app
  * (probably want to let the user know what)
endif
```

```
*  
* clean up and close shop  
*  
* upon termination, restore menu  
***pop menu _MSYSMENU to master  
set sysmenu to default  
* return environment  
oEnv.RestoreSets()  
* clean out everything else  
close all  
clear menu  
clear popup  
clear program  
clear window  
clear  
if m.gTalkIsOn  
    set talk on  
endif  
* from within VFP or from runtime  
if version(2) = 0  
    wait window nowait "See ya later, alligator"  
    clear memory  
    quit  
else  
    @2,0 say "See ya later, alligator"  
    clear memory  
    return  
endif  
  
* <EOF>
```

So what's the deal? This just looks more complex than our old IT.PRG, and certainly harder to track down and maintain, wouldn't you say? Well, here's the advantage. Suppose you just realized that you should be saving the setting of HOURS because some users have been messing with the setting of the clock and you need it to look just so in certain places.

In the olden days (about 10 paragraphs ago), you would have had to put another three lines in IT.PRG:

```
m.gcOldHour = set("HOUR")  
set hour to 12  
set hour to &gcOldHour
```

That's not too bad, is it? But you'd then have to repeat these three lines of code in every copy of IT.PRG in every application. And, then, finally, supply a new copy of your executable containing this new IT.PRG.

Yes, I'm sure that's how you'd like to spend one of your Saturdays.

If the thought doesn't appeal to you, here's the OOP way. First, you'd modify the ENV class by adding a new property, nOldHour. Then you'd change the SaveSets(), DoSets(), and RestoreSets() methods and include the above lines in the appropriate places. (This is left as an exercise to the reader. Don't you just love it when you see this line?)

And that's it. The next time you run the app with this class, the HOURS setting is automatically handled. Of course, you have to include the new .VCX with your app, but again,

it's just one change. And if you're still unsure, note that I also haven't touched on the possibility of subclassing the methods in ENV, for example. I'll get to that later.

## Making setting up and cleaning up more automatic

You've noticed that I've basically substituted a function call to a class method in place of a bunch of in-line code. I could have done the same thing with a procedure call, of course. And the same benefits of being able to make a change in one place would also have accrued. However, the potential benefits of subclassing notwithstanding, there's yet another benefit.

In the ENV class, I created three separate methods: SaveSets(), DoSets(), and RestoreSets(). But anyone who uses this class has to remember to call these methods. In this particular case it isn't a big deal, because you're probably only going to call each of them once, but it's pretty easy to imagine some examples of twin "setting up" and "shutting down" functions that would be called all the time. Two that come to mind immediately: opening and closing a table, for example, or using an ActiveX control.

You can put code in the Init() and Destroy() methods of a class, and that code will automatically be fired when the class is instantiated and when its object reference is released. So, instead of having a separate SaveSets() method, put that code in the Init() method, and then move the RestoreSets() code to the Destroy() of the ENV class. Then you could do this instead:

```
set classlib to BASEAPP.VCX
oEnv = createobject("ENV")
oEnv.DoSets()
*
* bunch of code here, including the event handler
*
release oEnv
```

When the ENV object is instantiated, the SaveSets() code will be automatically executed and those values stored to the properties of the ENV object. They will then hang around until you get rid of the oEnv object reference, which, again, is just a memory variable. So the RELEASE oEnv command will "un-instantiate" the ENV object and also fire the code in the Destroy()—the RestoreSets() code. An even more sophisticated method is to leave SaveSets() and RestoreSets() alone, but to call those methods from the Init() and Destroy()—so you can call them from other places in your application if needed.

I've still got a separate DoSets() method, because I might want to call that from elsewhere in the application.

Structuring your class code this way makes it easier to manage and use, and once you get in the habit of doing so, you'll wonder why you spent so much time the old way.

## Enhancing the non-visual class

 This business of setting up classes is a lot to handle, considering that in the beginning of this chapter, you hadn't started programming yet. It's a lot to absorb, and I almost felt it was too much to handle in the same breath as working with menus, the event handler, and so on. But it's really not that bad when you take it a step at a time, is it? How about another shot at it. The source code in question is in the Chapter 12C directory.

You might have noticed that I've objectified just one small piece of the setup program. Shouldn't it all be handled the same way? Well, that's not going to be too tough, since I can either add to the existing methods or create new methods to handle other pieces. The important thing to remember is that the class library has to be opened as soon as possible, so that the internals can be accessed immediately.

What I'm going to do is create a second class, APP, that also resides in BASEAPP.VCX, and move more of the setup code into methods of that class. Here's what IT.PRG looks like now (you'll notice that I also ripped out the code that handles the menu and sets up the READ EVENTS loop; more on that in a minute):

```
* it.prg
* top level program

* set stuff up
* talk is a special case
if set("talk") == "ON"
  set talk off
  m.gcOldTalk = "ON"
else
  m.gcOldTalk = "OFF"
endif

* clean out everything
release all except g*
close all
clear

* set up outside resources
set classlib to BASEAPP.VCX
oEnv = createobject("ENV")
oApp = createobject("APP")
* (stubbed out for this example)
*\\ set help to ("MYHELP")
set procedure to ("MYPROC")
*\\ set resource to ("MYRES")

* call the ReadIni method or set up initial
* parms as necessary
oApp.ReadIni()

* define environment
oEnv.DoSets()

* get the user
oApp.Login()

* set up event handler
oApp.it()

* return environment
* this releases all objects in memory
release all except g*

* clean out everything else
close all
clear menu
```

```
clear popup
clear program
clear window
clear
set talk &gcOldTalk

* from within VFP or from runtime
if version(2) = 0
  wait window nowait "See ya later, alligator"
  clear memory
  quit
else
  @2,0 say "See ya later, alligator"
  clear memory
  return
endif

* <EOF>
```

Especially considering that most of this is comments, that's quite an improvement over the old version, isn't it? And the beauty is that because all of the "work" is being handled inside classes, changes that need to be made will automatically be inherited by the apps that rely on those classes.

I've really done nothing conceptually new over the initial creation of the ENV class and the first couple of methods—all I did was take advantage of the Init() and Destroy() methods of the ENV class and move the SaveSets() and RestoreSets() code there as I talked about in the previous section.

But what about this oApp object?

## Creating an application-level non-visual class

If I was to leave IT.PRG as is, with a bunch of calls to the methods that belong to the oEnv object, it would feel kind of like I'm just half done. I talked about all of this object-oriented stuff to get the app ready, and then reverted back to procedural code in order to set up the menu and the event handler. What's wrong here? The APP class contains methods to set up the application itself. These methods will handle the menu and READ EVENTS mechanism, as well as other application-specific functions like opening data files, running a login script, handling the opening and manipulation of forms, and so on. I'm not going to cover all those pieces in this chapter, but I will address those that are needed to get the application running to the point that the menu is available.

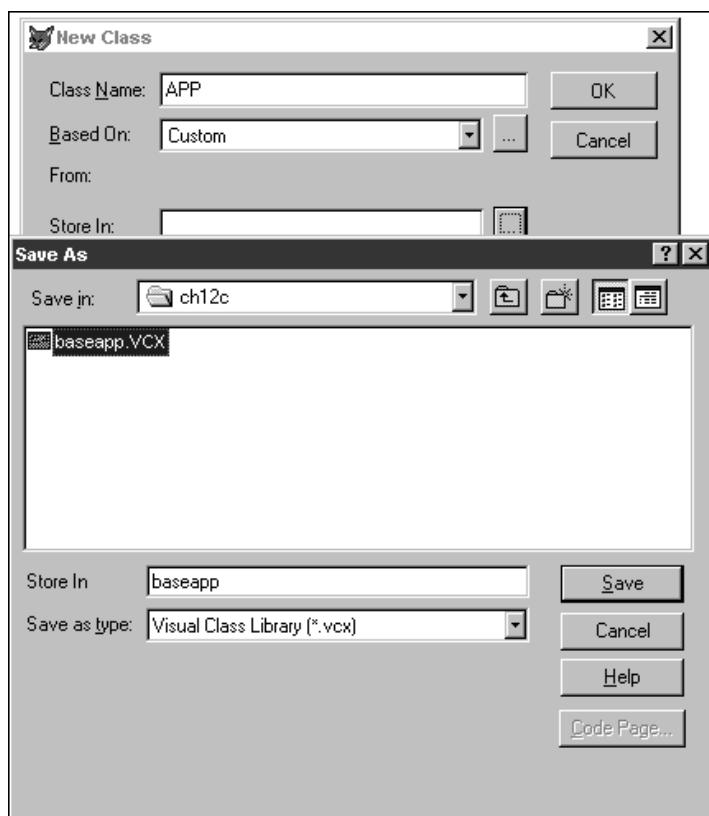
Here are the steps required to add the APP class to BASEAPP.VCX:

1. Create a new class by issuing the CREATE CLASS command, or selecting the File, New, Class menu option or selecting the Classes node in the Project Manager and clicking the New button.

2. Enter the following in the New Class dialog box:

- Class Name: APP
- Based On: hwcustom
- Store In: BASEAPP.VCX

Note that this time, you're adding the class to BASEAPP.VCX, which already exists, so click the ellipsis button beside the Store In text box, as shown in **Figure 12.12**. The Class Designer will appear and our funny-looking, three-color friend will appear in the Class Designer window.



**Figure 12.12.** Adding a class to an existing class library.

At this point, it's time to add methods and properties to the class. I'm going to add seven custom methods and use one existing method to store code as well. Here they are (I'll explain how they were created once I've laid them all out):

**oApp.Init()**

```
this.lCanGoOn = .t.
```

**oApp.readini()**

```
* initialize various system attributes
* by reading in from outside the application
* (either an .INI file or the Windows Registry)
This.cNameSystem = This.GetRegValue("NameSystem")
This.cVersion = This.GetRegValue("VersionSystem")
This.cDefaultDataLocation = This.GetRegValue("DefaultDataLocation")
This.nMinRAM = This.GetRegValue("MinRAM")
This.nMinDiskSpace = This.GetRegValue("MinDiskSpace")
* make sure the default data set exists and is good
if !This.DataIsGood(This.cDefaultDataLocation)
  *\\ in real system, we'd set this flag if the test failed
  * This.lCanGoOn = .f.
endif
* check the hardware and OS
* (each of these functions is in MYPROC)
This.nAvailRAM = This.GetRAM()
if This.nAvailRAM < This.nMinRAM
  *\\ in real system, we'd set this flag if the test failed
  * This.lCanGoOn = .f.
endif
This.nAvailDiskSpace = This.GetDiskSpace()
if This.nAvailDiskSpace < This.nMinDiskSpace
  *\\ in real system, we'd set this flag if the test failed
  * This.lCanGoOn = .f.
endif
```

**oApp.login()**

```
* initialize user login parms
This.cNameUser = "ADMIN"
This.cPassword = "ADMIN"
This.cPermLevel = "00001"
m.llLoginWasGood = .t.
* get user login
* user login returns true or false
do form LOGIN to m.llLoginWasGood
if !m.llLoginWasGood
  This.lCanGoOn = .f.
endif
```

**oApp.it()**

```
* save old menu, run new one
if This.lCanGoOn
  push menu _MSYSMENU
  do IT.MPR
```

```
read events
else
  * something bad happened so can't run app
  * (probably want to let the user know what)
endif
*
* clean up and close shop
*
* upon termination, restore menu
pop menu _MSYSMENU to master
set sysmenu to default

*\\\ these methods are placeholders
```

### **oApp.GetRegValue()**

```
lpara m.tcDummy
return .t.
```

### **oApp.GetDiskSpace()**

```
lpara m.tcDummy
return .t.
```

### **oApp.GetRAM()**

```
lpara m.tcDummy
return .t.
```

### **oApp.DataIsGood()**

```
lpara m.tcDummy
return .t.
```

As you read through these methods, you can see how they tie to the procedural code that made up IT.PRG. Remember that the Init() event is automatically fired when the oApp object is instantiated, so it's a good time to set the value of the lCanGoOn property that indicates whether or not various tests during set up are passed.

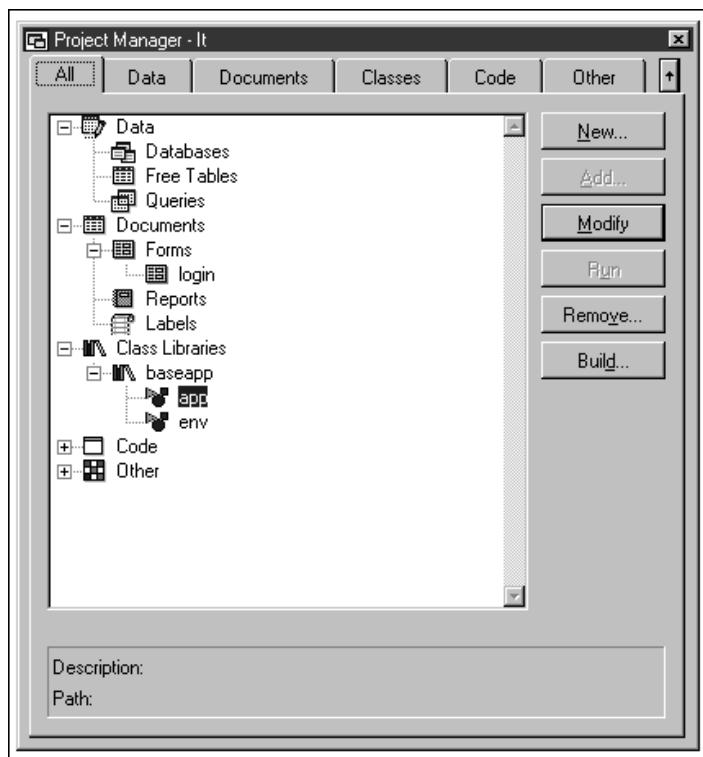
The ReadIni() method is moved into the App class so it can be called from anywhere in the app, and can also be subclassed should you need a different way to access an .INI file (such as if you wanted to read values from the Windows Registry). Similarly, the Login() method, which gets the username and password from the user, can also be called from anywhere in the app, and can also be subclassed if you wanted to use a different login screen, or if you wanted to bypass a login screen completely—if your app didn't need security, or if you read login information from the operating system.

Both of these methods have places to set oApp.lCanGoOn to False so that the app terminates before trying to set up the event handler.

The It() method (you knew I was going to call it that, didn't you?) is called from the heart of IT.PRG, and is used to set up the menu and run the READ EVENTS command.

You can also see a number of examples of calling methods from various places in the application, as well as referencing the values of and setting the values of properties of various objects. One of the most common errors when moving into object-oriented programming is forgetting the object! It's a rare programmer indeed that doesn't still forget the reference to "This" preceding the name of a property in the APP class.

Before I go on, I want to take a step back and review how APP and ENV and IT.PRG all fit into an application. If you're working with the Project Manager, you should be creating and modifying your programs, menus, and classes there as shown in **Figure 12.13**. As a result, the files will automatically be added as they are created. If you issue commands from the Command window, the components will be added to the project when the application runs.



**Figure 12.13.** Be sure to use the Project Manager to manage all components of a project.

This question might come up: Why are Login() and ReadIni() part of the APP class, instead of the ENV class? Good question, and the answer is: "Because I said so." In other words, there

isn't any hard and fast rule about how to create classes and what to put where, any more than there were a set of rules handed down from high regarding the construction of procedures. Of course, this doesn't mean you should create classes and randomly populate them; leave that for the amateurs. The reasoning I'm using about "what to put where" is along the lines of "where does it most naturally fit?" The Login and ReadIni processes, to me, are really more application-specific than environment-specific. You might choose differently—all applications will have a name and a data location, while it's certainly feasible to conceive of an app that doesn't have a requirement for a login process.

Some developers don't like the idea of an "all-encompassing" application-level object, preferring to create a number of smaller "service" objects. One object would be responsible for data handling, another for managing forms, a third for user login and permission handling, and so on. It's really up to you to decide how you want to do it. Don't worry if you spend a lot of time creating and organizing your set of objects, only to find out you don't like how they ended up, and you have to start all over again—most developers find out they don't like their first pass.

This is a good place to bring up the idea of subclassing the APP class. What if you had an application that had special requirements for, say, logging in? How would you handle that? There are a couple of different approaches.

One would be to subclass the entire APP class, perhaps calling it APPX, and putting it in its own .VCX that belongs just to this application. Remember, BASEAPP.VCX is a general-purpose class library that will go in the COMMON directory—it's just in the current directory in these examples so it's easier for you to work with. Then we'd handle the special situations in the subclass APPX, instead of adding to the APP class. Because you're subclassing APP, any changes you have to make to other parts of the class are still reflected in the APPX class, of course. However, the downside here is that you're essentially duplicating the entire APP class, which can be somewhat of a resource hog.

Another way would be to keep APP as it stands and simply override the Login class. In this situation, you'd do the same for your own classes as you do for the Visual FoxPro base classes: You subclass your own base classes, and then work with those in your application. This can be useful if you need to override or otherwise modify your base classes a lot, but it can be a lot of overhead if you don't need to do so that often. And if you find yourself modifying your own base classes a lot, perhaps that's a signal that you need to redesign them.

# Chapter 13

## Building a LAN Application

**So now you've got a menu. You can run that menu and select the various menu options, and generally nothing happens—unless, of course, you find one that generates an error. This is technically sound but functionally uninteresting. What you'd like to do is “hang some forms” on your menu. In other words, the first thing a user is liable to want to do is open a form and enter some data. In this chapter, I'm going to show you how to add forms to your menu (using your own base classes, of course) and build a set of common methods that you can reuse instead of having to recode them for each form.**

In the olden days (remember them? about six months ago?), you'd create a menu option for a screen (you called them “screens” in those simpler times), create the screen—consisting of data elements and some controls such as Next/Previous and Add/Save buttons—and you'd be all done.

With Visual FoxPro, you're going to do things a little differently. Your ultimate goal is to have the ability to open multiple forms on the screen, and have users be able to move between them as they desire. In fact, you're even going to give them the ability—in selected situations—to open multiple copies of the same form. And you're going to provide a common toolbar for all of these forms. This toolbar will have buttons for each of the options that we used to have on a specific form: Next, Previous, Add, and so on. As a result, the toolbar becomes part of the application; it's no longer part of a single form. Furthermore, because you're going to make our forms friendly to those “rodentially challenged” folk (as people who don't like mice refer to themselves), you'll have a Record menu with its corresponding keyboard shortcuts.

To handle all of this, you're going to need to build some application-wide functionality (to handle the toolbar and menu options) as well as a generic engine to handle all of the goings-on that will be required for multiple forms. If you've looked at Tastrade, you've seen that this is all built in, and that you can fill up many a sheet of paper—and go through the better part of a bottle of aspirin—trying to map the relationships between the programs, classes, and forms. It's not impossible; there's just an awful lot to comprehend. And remember, you're trying to learn a whole new set of tools at the same time you're working on the syntax and concepts.



This chapter is going to create an application foundation that will handle your menus, toolbars, and form opening and closings. But because there is so much to this topic, and the pieces are so intertwined, I'm going to take it a step at a time, like I did with menus a few pages ago. If you're the type who has to poke around with code while you're reading, you're going to like this chapter a lot. The source code that accompanies this chapter consists of a number of .ZIP files, each of which contains all of the code explained here, up to a certain step. They each start with “CH13” so it will be easy to follow along.

First, you're going to build a simple form using those base classes of forms and controls, and use some “old-fashioned” navigation controls—Next, Previous, Add, and Delete—placed right on the form. The purpose of this form is to show you how to build a “real” data-entry form with your own classes instead of Visual FoxPro base classes. You'll also put some simple code in the navigation buttons to get you acquainted with that mechanism. Remember how I said that

many developers are going to bag the “object-oriented” stuff and just create forms and controls from VFP’s base classes? I’m going to show you the benefits immediately, so you learn the right way from the start. By the time you’re done with this step, you’ll have created a project, built the main program and menu, added the environment and application classes from the last chapter, and added your form to the menu.

The second step will be to add the same navigation functionality to the menu that already exists with the controls on the form. In other words, your users will be able to use these forms both via the controls on the form as well as via menu shortcuts. When you’re done, you’ll have a functional application: You’ll be able to open multiple copies of the same form from the menu, and use both the menu and command buttons to navigate.

Next you’ll create and include a toolbar with your application. You’ll learn how to create the toolbar, and find out how to launch it automatically when the form is opened. And you’ll also learn how to make the toolbar control the form’s navigation functions. As soon as you’re done, however, you’ll run into several problems, and you’ll then fix them. You’ll get rid of the command buttons on the form (they’re still there, sadly enough), and properly synchronize the toolbar and menu option functions. You’ll also build an instance handler—a mechanism that keeps track of which forms are open and how many times they’ve been opened—and add the form’s name to the Window menu.

The fourth step will allow you to solve some of those data issues you’ve started worrying about. Up to this point, all you’ve done is simply read values out of the table and display them on the form. And in some cases—such as with primary keys and lookups—you’re going to want to display something other than raw data from the table. You’ll also create generic methods for adding, deleting, and saving data. Then I’ll show you how to make those generic routines extensible by providing hooks that you can use for custom code in specific instances.

Then it will be time to add more user functionality to the form. I’ll show you how to add controls to handle child records as well as how to call child forms for both querying and for adding new records.

The final step in this chapter will be to tie up some loose ends. During the previous steps, I’ve intentionally left some rough edges, and here I’ll show you how to clean those up and integrate the components a bit more tightly.

At this point, you’ll have the rough beginnings of an application foundation that you can use to build your own applications. It won’t be finished (since when are we ever done with an application?) but you’ll understand how it was built. Then it will be time to create a couple of other “real-world” forms, and those are next.

## **More on the sample application**

Let me discuss the actual forms you’ll be creating, so that you understand where you’re heading. I’ve added a small DBC and some corresponding tables to it that represent the core data in this example. The first table is CUST, and, logically, contains a record for every customer in the system. The second table, ORDERH, contains a record for every order in the system. The third table, ORDERD, contains a record for every detail record in an order.

I’ll be using two forms as samples: one for invoices and one for orders. In both cases, of course, I’ll have customers attached to the form as well. The top half of each form will contain the customer info. The bottom half will contain information about orders and order details.

Two other tables will be used in this example: ITKEY, which contains the last primary key used for every table in the application, and ITLOOK, which is a generic lookup table. Obviously, ITKEY is needed when new records are added to CUST, ORDERH, and ORDERD so that those records can get new primary keys. I'll get to ITLOOK later in this chapter.

One other note and then I'll get started. Remember that long-winded explanation about setting up directories so that each subdirectory held various components? And have you noticed that, so far, I've completely ignored that advice and just slammed everything into one directory? And that I'm continuing to do so? Well, the reason is that while you're learning the basics, there simply aren't that many files to deal with. Furthermore, the mechanism to handle multiple-directory setup is complex enough that it will just get in the way.

Finally, during the development of these examples, you'll be making a number of changes to your base classes, so you will need separate copies for each set of examples. Instead of putting each version into the same common directory and then using some hairy naming convention to find them, I'm just going to put them in with the custom code for the example in question. That's why you see copies of the class libraries in every directory.

## Types of forms

Before I get into the business of building forms, it would be a good idea to discuss the types of forms you might want to build. In general, you can think of a form as any visual mechanism used by the user to handle data, issue instructions to the system, or respond to a message that the system issued. The data-handling forms can be further categorized into simple maintenance forms that address one table (and possibly some lookups), parent-child forms (often depicting a one-to-many relationship on the form), complex data-entry forms that have complex relationships between the various tables (and that rely on business rules for the definition of those relationships), and query-style forms that are used primarily to extract data instead of maintain it.

In this application, you'll find at least one of most of these form types. The INVOICE form is a simple maintenance form, while the ORDER form is a parent-child form. The INVENTORY form is an example of a query-style form.

And of course, we're going to have other types of forms, such as those for user logins, system management, and various alerts and warning messages.

Just like menus, you could create forms using the Form Designer and then grab controls from the Form Controls Toolbar, and that's likely what many are going to do. They're not going to see the benefit of subclassing those controls—only the extra work involved in creating and using them to create their forms. However, six months or a year down the road, they're going to have a number of applications put together, all using the base controls, and they'll be chin-deep in a maintenance nightmare. And then their customer or boss is going to tell them that all of the magazines and books say how easy it is to change an application because of Visual FoxPro's object-oriented capabilities ...

So start off on the right foot right away, and build your own base classes for forms and controls. By the time you're done with this chapter, you should be completely sold on the benefits of using Visual FoxPro's object model.

## Setting the stage: creating your class libraries

As you remember from the last chapter, a class is a set of one or more records that describes either a visual object, such as a form or a control, or a non-visual object, such as a set of methods and properties. A class can live in its own table, or it can coexist with other classes in the same table. This table is called a *class library*, regardless of how many classes are in it.



You're going to create one class library that will contain both form classes and control classes. In the first version of this book, I had two separate class libraries—one for forms and one for controls. I never used one without the other, so I decided to put everything into one library, called HWCTRL. (Actually, it's called HWCTRL62.VCX, so that I don't accidentally confuse it with a development version of a class library on my machine.) This library will contain the definitions for your own base classes. These classes include a form subclassed from Visual FoxPro's base form and controls subclassed from Visual FoxPro's base classes. You'll use the form class (hwfrm) to create several types of forms—one for simple maintenance (hwfrm), another for parent child forms (hwfrmPC), and so on. Similarly, you'll create control classes (hwtxt, for example) and use them to create additional types of controls—one for read-only fields (hwtxtRO) and another for fields that should be displayed only if a developer is running the form (hwtxtDev).

Some people organize these classes somewhat differently. As you saw, my text box class, hwtxt, is subclassed directly from Visual FoxPro's base class, but I use it for normal data entry. If I need specialized versions of it, I subclass hwtxt into hwtxtRO (read-only) and hwtxtDev (developer-visible).

Other developers would never use hwtxt directly. Instead, they'd use hwtxt strictly as an abstract class. Hwtxt would spawn three classes instead of two: hwtxtEdit for a data-entry text box, as well as hwtxtRO and hwtxtDev.

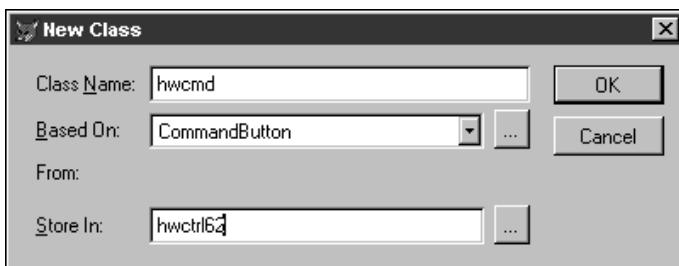
The important thing to remember is that you've got a two-level hierarchy here. hwFrm comes from Visual FoxPro's base class, while hwfrmMaint and hwfrmPC are both subclassed from hwfrm. You'll use hwfrmMaint and hwfrmPC to create the actual forms for your applications. Then, if you decide you've done something stupid, you can go back to one of your subclassed forms, make a change to the class, and have that change ripple through the class hierarchy.

## Creating HWCTRL62.VCX, your control class library

If you're getting sick of the "step by step" spoon feeding we're going through as we build these classes, please bear with me. I'd rather err on the side of conservatism and repeat steps that you don't need, instead of breezing by something to the tune of "And you just build a rocket ship, and then you can..." Meanwhile, you'd be saying, "Whoa, let's take that 'Build a rocket ship' step a bit more slowly."

### Create your controls class library

1. Issue the CREATE CLASS command or select the File, New, Class menu option.
2. The New Class dialog appears. Enter the following information as shown in **Figure 13.1**:
  - Class Name: hwcmd
  - Based On: select CommandButton
  - Store In: HWCTRL62.VCX



**Figure 13.1.** Creating the hwcmd class based on Visual FoxPro's CommandButton base class.

Note that “hwcmd” will be a record in the HWCTRL62.VCX table. In OOP terms, hwcmd is a class contained in the HWCTRL62 class library. You don’t have to include the .VCX extension, but I did it here to make it clear what was getting stored in what.

3. The Class Designer appears with a command button in it.

A couple of things are worthy of mention here. First, the window inside the Class Designer (not the Class Designer window itself) that contains the command button is just big enough to hold the command button. A lot of people don’t like this, and their first move is to resize the window, as shown in the two images in **Figure 13.2**. When they do so, however, the command button also resizes. In other words, the size of the window in the Class Designer defines the size of the control. Second, now that I’ve told you that you can’t resize the window inside the Class Designer, it’s only fair to note that you can resize the Class Designer window itself to be significantly smaller if you need more screen real estate.

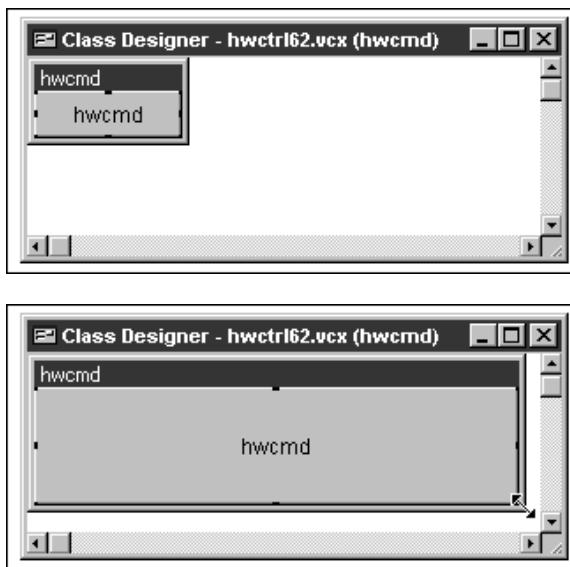
4. Make changes to the control as you desire.

Remember, this control (a command button in the current example) will be the one from which all future controls are created—both controls on forms as well as future subclasses of buttons. If you want the Click() method of all your command buttons to move the cursor position to the beginning of the text in the text box, you change this

control's Click() method, and all command buttons you create from this one will automatically inherit that behavior.

Because this is your first shot with the various controls, the list below this set of instructions describes each control whose subclasses we're going to modify. The rest will be subclassed but left alone.

In this example with the command button, change the caption to "hwcmd" and the size to 23 pixels high and 72 pixels wide. Why? I personally like this particular size as a default size, and I typically change the caption for each of my subclassed controls so if I accidentally put a Visual FoxPro control on a form, I'll see the caption "Caption" (instead of the caption "hwcmd") and it clues me in that I wasn't thinking straight.



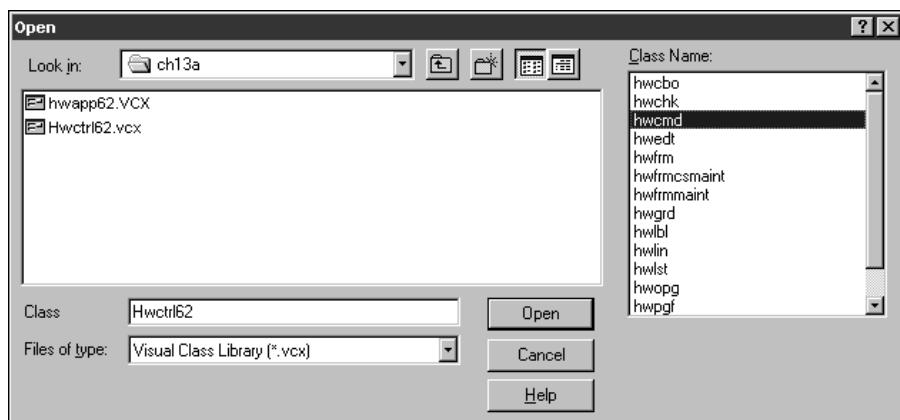
**Figure 13.2.** If you resize the object inside the Class Designer window, you'll actually be changing the size of the control.

5. When you're done, select the File, Save menu option.
6. Select an icon for this class from the Form Control toolbar.

As you may remember, in order to use your own controls, you need to register the library. Once you've done so, the Form Control toolbar displays all of the controls in that library in place of Visual FoxPro's controls. Visual FoxPro will automatically assign an icon based on the base class you selected in the New Class dialog. If you want your own icon instead, here's how to select the icon image:

- Select the Class, Class Info menu option. (The Class Designer window must be active for the Class menu pad to be available.)

- Select the name of the file that contains the icon to be displayed in the Form Control toolbar, and place it in the Toolbar Icon text box.
  - You can also enter a description for this class. If you are creating a number of similar classes, you might find that this description comes in very handy later.
  - Click OK.
7. If you need to change properties, use the Class Designer:
- Issue the command MODIFY CLASS.
  - Highlight the name of the desired class library (.VCX) in the list box on the left side of the dialog box. The classes in that class library will appear in the list box on the right side of the dialog box as shown in **Figure 13.3**.



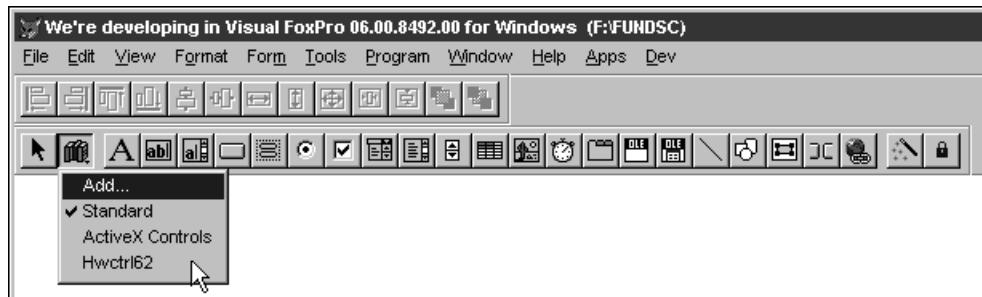
**Figure 13.3** The Class Name list box contains a list of all classes in the selected .VCX.

- Select the desired class. The Class Designer appears.
  - Make changes to the properties as needed and save them when finished.
8. If you need to rename or delete classes, use the Class Browser:
- Open the Class Browser.
  - Use the Remove or Rename icons as required.

If you're following along on the computer, you can create your own classes as you desire—you might want to use the suggestions for classes in Chapter 10.

### Register the library

When you click the View Classes button (the one with the books) in the Form Controls toolbar, you'll see a list of all registered classes, as shown in **Figure 13.4**.



**Figure 13.4.** Registered class libraries appear in the Form Controls toolbar drop-down menu.

In order to add HWCTRL62.VCX to that list, you need to register it with Visual FoxPro using the Tools, Options dialog. Follow these steps:

1. Select the Tools, Options menu option.
2. Select the Controls tab.
3. Click the Add command button to add HWCTRL62.VCX to the Selected list box.
4. Click the Set as Default command button to keep this class library registered between sessions of Visual FoxPro.
5. When you select the View Classes icon in the Form Controls toolbar, your newly registered class library will show up in the list and you can select it to populate the toolbar with your own subclassed controls.

If you want to just add a class for the current session of Visual FoxPro, you can simply select the Add menu option from the Form Controls' View Classes button, and then select the desired class library from the Open dialog that appears.

### Creating your form classes

You had to create your controls first because you're going to use a couple of them when creating your form base classes.

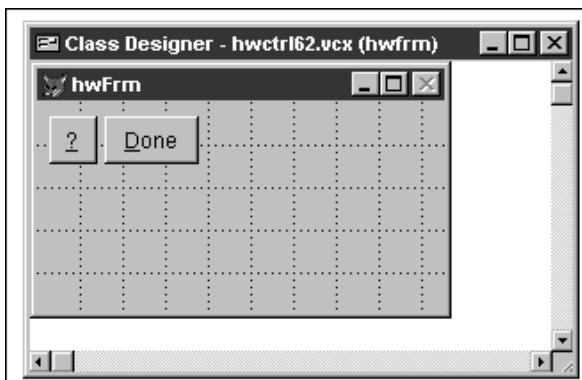
#### Create your form base class

1. Issue the CREATE CLASS command or select the File, New, Class menu option.
2. The New Class dialog appears. Enter the following information:

- Class Name: hwfrm
- Based On: select Form
- Store In: HWCTRL62.VCX

Note that “hwfrm” will be a record in the HWCTRL62.VCX table. Again, in OOP terminology, “hwfrm” is a class contained in the HWCTRL62 class library.

3. The Class Designer appears with a gray form in it. Make the following change:
  - Caption: hwfrm
4. Add two command buttons as shown in **Figure 13.5**. I always have a Help button and a Done button on my forms, so I put them right here. For the rare form that doesn’t need these buttons, it’s easy enough to turn their Visible property to False.



**Figure 13.5.** Add a Help command button and a Done command button to your form base class.

5. In the Click() method of the Done command button, add the following code:

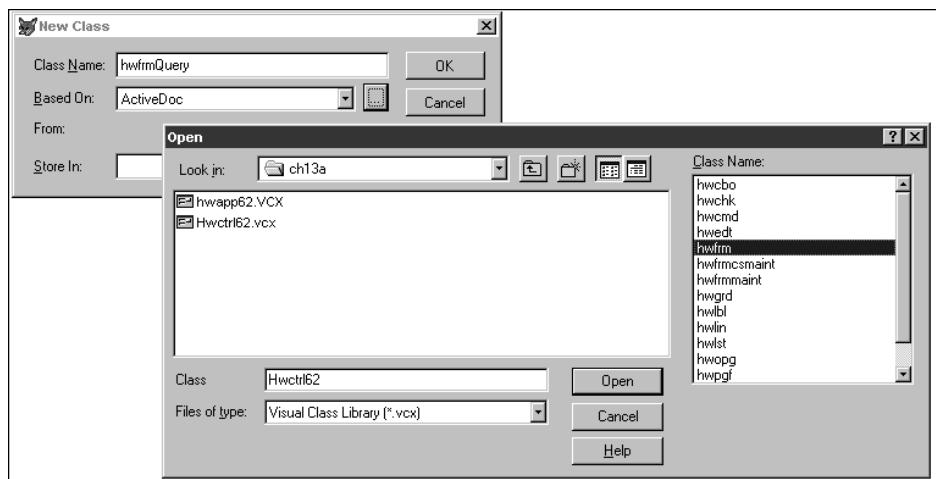
```
thisform.release()
```
6. Change the AutoCenter property of the form class to True and change the caption of the form to “hwFrm” so that you can easily identify which class this form was created from.
7. Save your class, and you’re done.

Here’s where the difference between the hierarchy of your control classes and form classes lies. The form you’ve just created is your base form, from which all future forms will be created. You’ll create a maintenance form, a parent/child form, and so on—all using this class, hwfrm, as the source. Then, if you decide you need to change any properties of your forms, you can do so here. You still get to do many variations on a theme by subclassing hwfrm.

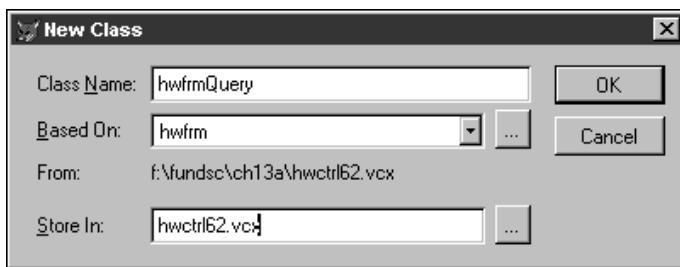
## Create your maintenance form class

1. Issue the CREATE CLASS command or select the File, New, Class menu option.
2. The New Class dialog appears. Enter the following information:
  - Class Name: hwfrmQuery
  - Based On: ActiveDoc
  - Store In: HWCTRL62.VCX

Stuck yet? I tried for months trying to figure out how to enter my own class name in the Based On drop-down list box. Well, maybe not months, but it sure was aggravating. It's one of those "little tricks" that isn't immediately obvious, but once you've learned the trick, you're more than willing to show someone else. Click the ellipsis (the three dots) command button to bring forward a list of available class libraries, and select "hwfrm" from the HWCTRL62 class library. In **Figure 13.6**, I'm creating another form class, hwfrmQuery, also based on the hwfrm class.



**Figure 13.6.** Use the Based On ellipsis command button to bring forward the Open dialog, from which you'll select the hwfrm class from the HWCTRL62.VCX class.

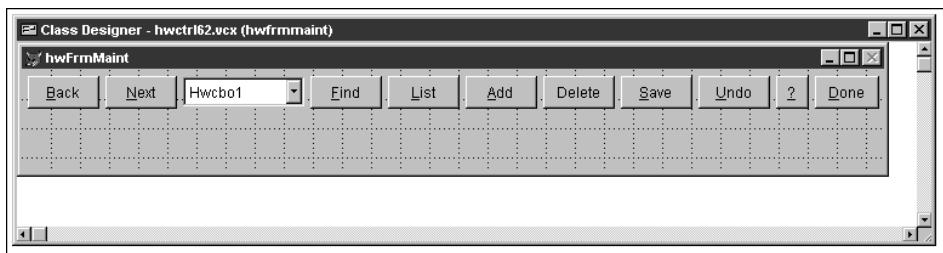


**Figure 13.7.** The New Class dialog box should look like this once you've selected the correct class and class library.

3. The Class Designer appears with a gray form that has a caption property of "hwfrm". You'll notice your Help and Done command buttons as well.

The class "hwfrmMaint" is the start of your base form for standard data entry and maintenance. You'll add some controls to the form and then place code in the various methods attached to this form. This collection of controls and methods will become your default maintenance form, which I'll refer to during the rest of the chapter.

4. Change the caption of the form to "hwFrmMaint".
5. Add command buttons and a combo box to the form class so that it looks like **Figure 13.8**.



**Figure 13.8.** The hwFrmMaint form class has standard navigation and maintenance command buttons.

Be sure to add command buttons from the HWCTRL62 controls toolbar, not Visual FoxPro's Form Controls toolbar. You'll be able to tell immediately because the buttons will have "hwcmd" as their caption!

6. Notice that the Delete command button doesn't have a hotkey—you don't want to make it too easy for a user to hit the Delete key by accident. Align the command buttons using the controls in the Layout toolbar and Format menu. The Align Left Edge toolbar button will do what you think it will do. Most likely, however, the buttons will still have uneven spacing between them. Use the Format, Vertical Spacing, Make Equal menu option to make the spacing the same between each control.

Visual FoxPro will use the space between the first two buttons as the default space. Note that the toolbar buttons and the menu options are not enabled until you have selected a group of objects on the form.

7. You'll notice that the command buttons have names (not captions) of "hwcmd1", "hwcmd2", and so on. Rename each of the command buttons with better names, like so: "hwcmdBack", "hwcmdNext", "hwcmdAdd", and so on.
8. Add the following code to the appropriate methods:

```
hwcmdBack.click()
if !bof()
  skip -1
  _screen.activeform.refresh()
endif

hwcmdNext.click()
if !eof()
  skip
  if eof()
    skip -1
  endif
  _screen.activeform.refresh()
endif

hwcmdFind.click()
messagebox("You have pressed the Find command button")

hwcmdList.click()
messagebox("You have pressed the List command button")

hwcmdAdd.click()
messagebox("You have pressed the Add command button")

hwcmdDelete.click()
messagebox("You have pressed the Delete command button")

hwcmdSave.click()
=tableupdate()
_screen.activeform.refresh()

hwcmdUndo.click()
=tablerevert()
_screen.activeform.refresh()
```

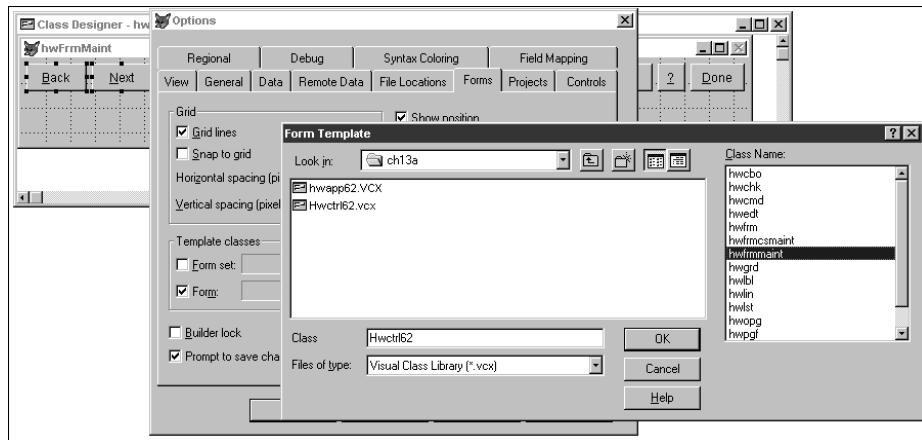
Note that you don't have to add code to the Done button because the code was already put into its Click() method in the hwFrm class.

### **Set the form template class**

Now that you've got a maintenance form class, you can use it to create real live data-entry forms. Remember Chapter 8 on the Form Designer? You'll need to set the form template class in the Options dialog box:

1. Select the Tools, Options menu option.

2. Select the Form tab.
3. Select the HWCTRL62 class library and the hwfrmMaint class as the Form Template Class (in the bottom half of the Forms tab) as shown in **Figure 13.9**.
4. Click OK. Don't Save As Default unless you want to continue using this class across sessions of Visual FoxPro.



**Figure 13.9.** Don't forget to select the maintenance form class as the Form Template Class before creating your actual data-entry forms.

## Creating a real form from your base class

Now it's time to cook! You've got a form class and a number of control classes. The first operation to try is to create a simple maintenance form—one that allows you to maintain customers and their orders—as shown in the bottom of **Figure 13.10**. However, I'm not going to have you do the whole form at once. For the time being, you'll just do the part that has to do with a customer, as shown in the top of Figure 13.10. Later in this chapter, you'll add the orders and line-items sections of the form.

### Create the ORDERS maintenance form

1. Create a new form by using the CREATE FORM ORD command (or select the File, New, Form menu option.) The actual name on disk will be “ORD”.
2. The Form Designer appears with the caption ORD.SCX in the window’s title bar.  
Note that you already have a form with a bunch of controls on it. Not bad for one command, eh?
3. Change the caption of the form to “Order Maintenance.”
4. Change the name of the form to “ORD.”

**Form Designer - ord.scx**

**Order Maintenance**

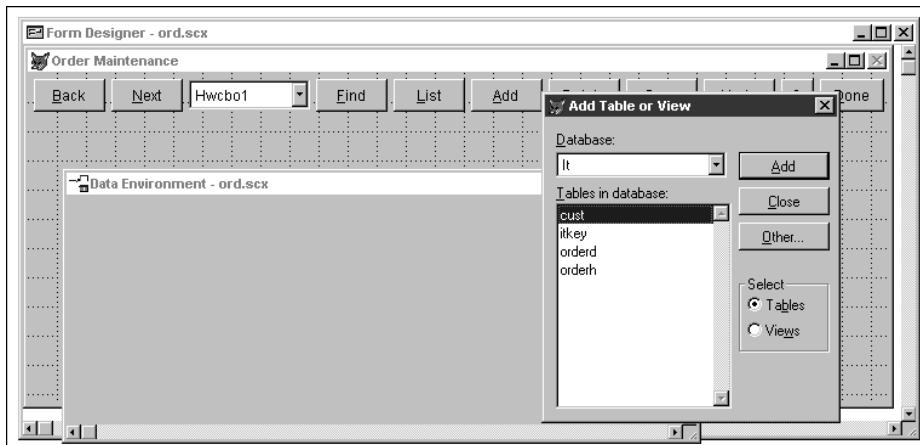
		Back	Next	Hwcb01	Find	List	Add	Delete	Save	Undo	?	Done
First	hwtxtFirst	Middle	hwtxtMiddle	Last	hwtxtLast							
Company	hwtxtCompany											
Address.1	hwtxtAddress1			Address.2	hwtxtAddress2							
City	hwtxtCity	State	hwtxtState	Zip/Postal	hwtxtZip	Country	hwtxtCountry					
Voice	hwtxtVoice	Fax	hwtxtFax									
Email	hwtxtEmail	Source of Customer			hwtxtSource							
Gossip	hweditGossip						<input type="checkbox"/> Want Access to Downloads					
							Username	hwtxtUsername				
							Password	hwtxtPassword				
Order#	Date	CC #		Exp.	Name On Card	Amt Charged	Date Charged	Auth #				
<b>hwlstOrders</b>												
Order# Line# Item Unit List Disc % Qty Extended Shipping Total Date Shipped												
<b>hwlstItems</b>												

Figure 13.10. The Order Maintenance form after adding Customer controls.

5. Create a data environment for the form.

The data environment describes the tables, indexes, and relations that are available to support this form. When you run the form, the data environment is automatically opened, and the tables and relations can populate the form's controls.

- Select Data Environment from the Form Designer's shortcut menu or select the View, Data Environment menu option.
- Select the Add Table menu option from the data environment's shortcut menu or the Data Environment menu pad.
- Select the CUST table as shown in **Figure 13.11**.



**Figure 13.11.** Add the CUST table from the IT database to the data environment for ORD.SCX.

6. Add fields to the form from the data environment.



You can (but don't do it right now!) drag fields from the data environment to the form and the appropriate controls will be created on the form—text boxes for most data types, edit boxes for memo fields, and check boxes for logical fields. These controls will automatically be tied to their respective fields in the data environment. The ControlSource property of a control specifies which field in a table will be used to supply data values to that control. However, the controls that you drag onto the form are created straight from Visual FoxPro's base classes unless you map field types to your own classes. I'll discuss how to do this later in this chapter. So for the time being, I'll ask you to put fields and labels on the form the hard way. You'll create controls from the Form Control toolbar that contains your custom controls, and use the data environment to visually remind yourself as to which fields you're working on.

- Add the text boxes, labels, check boxes, and edit boxes to the ORD form as shown in the top half of Figure 13.10. Notice that the controls automatically inherit the physical attributes of your own subclassed controls.
- Change the properties of each control to reflect the control source for that control. If you have the Data Environment window open at the same time, you can visually inspect the list of fields to remind you which field ties to which control.

You can change the same property of a series of controls easily by selecting a control with the Object drop-down list box, changing the value of the property in the list box, and then using the Object drop-down list box to select the next control. The same property will be highlighted in the list box for the new control, and you can immediately start typing. When you are selecting a ControlSource property, the combo box contains the fields from the data environment.

You can also select a control, select the Control Source row in the Properties Window, and double-click. As you move from one control to the next, the next field in the data environment's table will be displayed.

7. Change the following properties of the form:
  - BufferMode: 1-Pessimistic
  - DataSession: 2-Private
8. Save the form. Name the file ORD if you haven't already given it a name.
9. Run the form by right-clicking on it and selecting the Run menu command.

If you've followed all the instructions, you will be able to move from control to control, edit data in existing records, and either save or undo your changes. Clicking the Find, List, Add, and Delete command buttons will display the appropriate message.

Yes, it's just that easy ...

## **What's happening behind the scenes**

When you run the form, the form knows that there is a data environment attached to it (it's now a property of the form). It opens the tables in the data environment and sets any relations that are defined. When you close the form, the tables in the data environment are automatically closed as well. You can verify that this is going on by opening the Data Session window and changing the contents of the Current Session drop-down list box.

When you click one of the command buttons, you are executing the code in the Click method of that button. As you move between buttons, that much seems obvious. However, the constant calls to the \_screen.activeform.refresh() method might not be as intuitive. First, the Refresh method moves data from the table to the form's controls. If you don't call the Refresh method, clicking the Next or Previous command button will move the record point to a new record, but the form will continue to display the old data. Comment out the Refresh line in one of the methods and watch the record number change in the status bar while the form's display stays the same.

Second, what's the deal with this “`_screen.activeform`” syntax? Why not just use the name of the form? The reason is because this code is contained in a form class—and so it could be called by a form named anything. Because you need to reference the current object (the current form), but you don't know the current object's name, this roundabout method references the current form's Refresh method.

## Inheritance at work!

If you look carefully, you'll see that the two screen shots in Figure 13.10 show that I didn't follow my own advice earlier about renaming the controls in the `hwfrmMaint` class. You'll see that the combo box still has the default name, “`hwcbo1`”, instead of the name I should have given it—`hwcboSort`. This is not a problem, though. I just closed `ORD.SCX`, opened the `hwfrmMaint` class, and named the controls on the form properly. (You can't have a form that relies on a class—or subclass of a class—open when you want to open the class.) Then, when I reopened `ORD.SCX` again, the controls in this specific form also had the correct names. This is inheritance at work.

Of course, I didn't have to name the Help and Done buttons in the `hwfrmMaint` class, and, in fact, I couldn't even if I wanted to. The Name property for these two command buttons is dimmed and italicized in the `hwfrmMaint` class because the buttons are part of the class.

You can experiment yourself. Try changing properties of the controls in the class, or adding more code to the buttons—and then run the Order Maintenance form again to see the properties and behavior inherited by the form.

## Thinking about more application modifications

This is so slick! The first few times I did this, I was so excited—I didn't even know what I would use all this power for, but I knew it was going to be good for something. And you might be feeling a little cocky now, too. But wait! I'd like to point out a few, er, problems with this.

You see, I bet what you'd really like is one of those fancy toolbars instead of chunky buttons on the form itself. So what's the big deal? Just create a toolbar, copy the code that was in the Click methods of the command button into the Click methods of the various toolbar buttons, and you're all set, right?

Not right! First, do you really want to duplicate that code? Whenever I hear the word “copy”, the hair on the back of my neck stands on end. It doesn't sound very productive to me—and if you're going to have a toolbar, you're going to want it for all your forms. And that means that different forms will probably do different things. In other words, the Save method or the Next method will need to perform different actions according to which form you're on.

So where does the code go? Well, to make a long story short, the code does not get attached to the toolbar icon. The toolbar icon is just a mechanism to fire a method—it does not contain the method itself. But think for a minute. That means that the command button shouldn't have contained the method either. It should have contained a mechanism to fire the method, which is sitting somewhere else. And that means that you could even use a menu option as the mechanism to fire the method. This is getting good, isn't it?

So again I ask, where does the code go? What you should have is a call to some sort of magical “`next()`” method in the menu option, the toolbar, or the command button—but not the code for the `Next()` method itself. Eureka! The `Next()` method should be in the form! Well,

more precisely, it should be in the class—not the form itself. Just as you've subclassed the frmBaseMaint form and automatically had a Help button and a Done button come along for the ride, the methods of the hwfrmMaint class can come along. And if you need to, you can “enhance” those default methods—either by overriding them or by doing additional things. I'll show you both ways.

Let's try this again...

# Chapter 14

## Your First LAN Application: Building Upon Your Forms

So now you've got not only a menu but also a set of classes for forms and controls, and even a couple of forms based on those classes. In this chapter, I'm going to show you how to integrate those simple forms with the menu and add toolbars. Then I'll show you how to create generic functions for common tasks like Add() and Save(), and place those into your classes. This chapter is where it gets really good, and you start to see the power and flexibility of object-oriented programming.

This chapter has two parts to it. In the first part I'll show you how to make your forms, menus and toolbars all work in concert. Instead of keeping redundant copies of functions in various places, you'll learn how to consolidate code in a generic method. Once you've gotten comfortable with that, it'll be time to enhance your form classes with generic method calls—the classic “write once, use many” style of programming.

### Enhancing your form class with toolbars and menus

There's a bit of housekeeping to do before adding toolbars and menus; as you add these components to your application, you'll find you don't want to keep a bunch of files around willy-nilly. You already used the Project Manager in the previous chapter to build your menu—now it's time to resurrect the Project Manager to enhance the menu as well as add more components to the application.



At this point, we'll be working with the code in the CH14A directory of this book's source code files.

### Putting it all into a project

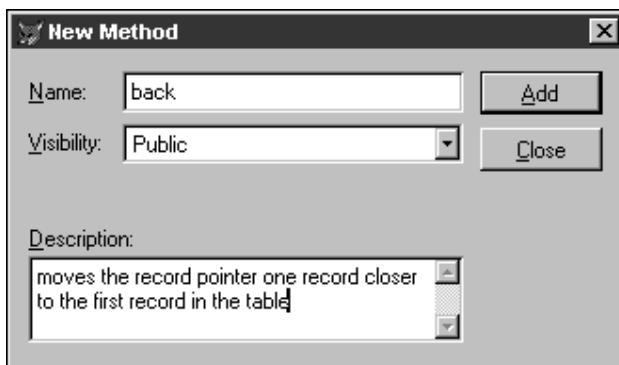
If your various files are scattered here and there, you'll probably want to create a new directory and copy all of the files there. Then open the project (or create one, if you didn't bother with that in the last chapter) and rebuild it. You'll most likely find that it's a lot easier to modify a class definition by clicking its name in the Classes tab than to find the Command window (it's around here, somewhere, I just saw it a second ago...), issue a command, select the right class, and so on.

It's now time to move the code from the Click() methods in the various buttons to generic methods in the form.

### Creating a form with its own methods

First you need to modify your form base class and add methods like Next() and Save() to it. Then you're going to add menu options to the menu that also fired those methods, and build a new form from your new form base class. Here's how:

1. Modify hwfrmMaint of HWCTRL62.
  - Issue the MODIFY CLASS command and select the hwfrmMaint class from HWCTRL62.VCX.
  - Select the Add Method menu option from the Class menu pad.
  - Enter “back” as the name of the method and an appropriate description in the New Method dialog box. See **Figure 14.1**.



**Figure 14.1.** Adding the Back() method to the hwfrmMaint class using the New Method dialog box.

- Click the Add button, and you'll see that the dialog stays open but the contents of all of the controls are cleared. So this form performs a “Save and Add Another” operation. Add methods for Next, Find, List, Add, Delete, Save, and Undo.
2. Add code to the newly added methods, just as you did for the command buttons that the code is replacing.
  - Select the new method in the Properties window and double-click to open the Code window.
  - Enter code in the newly added methods as shown in **Figure 14.2**:

```
* hwfrmMaint.back()
if !bof()
    skip -1
    screen.activeform.refresh()
endif

* hwfrmMaint.hwcmbNext.click()
if !eof()
    skip
    if eof()
        skip -1
    endif
    screen.activeform.refresh()
endif
```

```

* hwfrmMaint.find()
messagebox("You have chosen the Find method")

* hwfrmMaint.list()
messagebox("You have chosen the List method")

* hwfrmMaint.add()
messagebox("You have chosen the Add method")

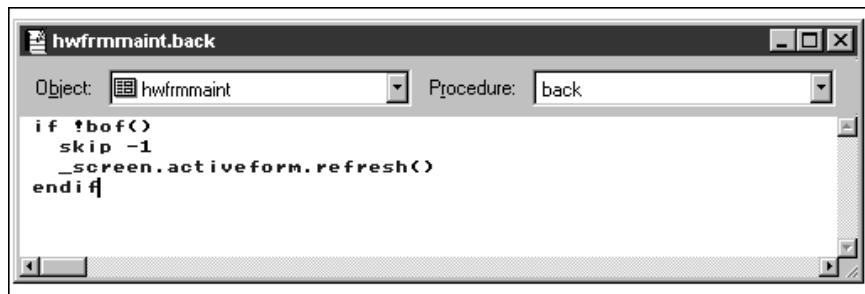
* hwfrmMaint.delete()
messagebox("You have chosen the Delete method")

* hwfrmMaint.save()
=tableupdate()
_screen.activeform.refresh()

* hwfrmMaint.undo()
=tablerevert()
_screen.activeform.refresh()

```

- When finished, save the class by using the File, Save menu option.



**Figure 14.2.** Use the Code window to enter code into newly added methods.

3. Create menu options for each method.
  - Make a copy of the IT menu from the previous chapter if you don't already have one.
  - Add the following commands to the corresponding menu options as shown in **Figure 14.3**:

```

Prompt: Back
Result: Command
_screen.activeform.back()
Option, Shortcut, Key Label: CTRL+PGUP
Option, Shortcut, Key Text: Ctrl+PgUp

Prompt: Next
Result: Command
_screen.activeform.next()
Option, Shortcut, Key Label: CTRL+PGDN
Option, Shortcut, Key Text: Ctrl+PgDn

```

```
Prompt: Find
Result: Command
_screen.activeform.find()
Option, Shortcut, Key Label: CTRL+F
Option, Shortcut, Key Text: Ctrl+F

Prompt: List
Result: Command
_screen.activeform.list()
Option, Shortcut, Key Label: CTRL+L
Option, Shortcut, Key Text: Ctrl+L

Prompt: Add
Result: Command
_screen.activeform.add()
Option, Shortcut, Key Label: CTRL+A
Option, Shortcut, Key Text: Ctrl+A

Prompt: Delete
Result: Command
_screen.activeform.delete()
(No shortcut for delete!)

Prompt: Save
Result: Command
_screen.activeform.save()
Option, Shortcut, Key Label: CTRL+S
Option, Shortcut, Key Text: Ctrl+S

Prompt: Undo
Result: Command
_screen.activeform.undo()
Option, Shortcut, Key Label: CTRL+R
Option, Shortcut, Key Text: Ctrl+R
```

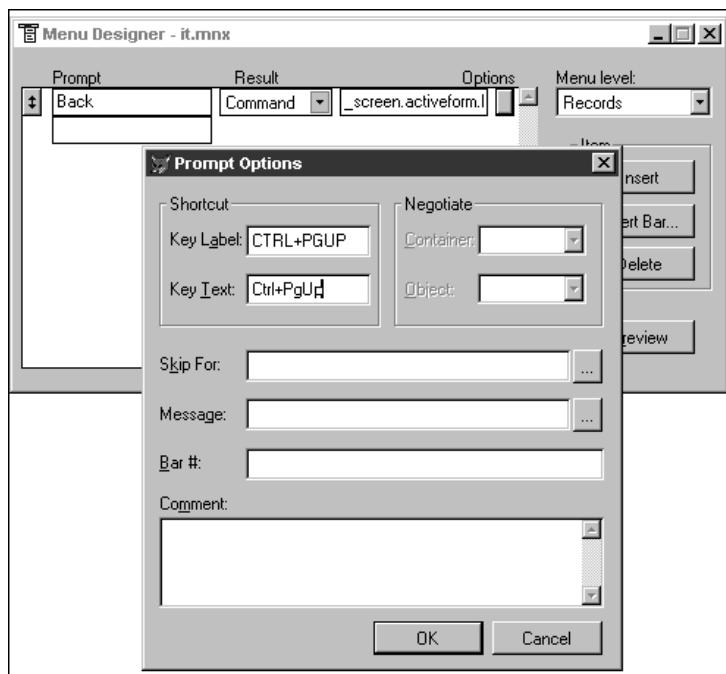
- Save the menu by using the File, Save command.

Note that now this same menu can be used for every form with a Next() method. There will be a bit of trickery involved in making sure that this menu option is okay to use on all forms, but that will come later.

## Creating a new form based on the new form class

Now that you've got the new form class ready as well as the updated menu, it's time to build a new version of the Order Maintenance form.

1. Modify hwfrmMaint of HWCTRL62. Don't forget to select the hwfrmMaint class as your template class and to register your HWCTRL62 class library.
  - Select the Tools, Options menu option.
  - Select the Form tab.
  - Select the HWCTRL62 class library and the hwfrmMaint class as the Form Template Class (in the bottom half of the Forms tab).



**Figure 14.3.** Add calls to the methods just created to the corresponding menu options in the IT menu.

- Select the Control tab.
  - Add HWCTRL62.VCX by clicking the Add button.
  - You can add your control class and Save As Default, and then set your Form Template class and click OK so that you don't use this class as your template for all new forms.
2. Create a new form with the CREATE FORM ORD command (or select the File, New, Form menu option).

 There's actually a quicker way to create a form from a specific base class, if you know the name of the class and the class library it's in. (I just didn't bring it up in the last section because there's only so much I can cover in one sitting.) Use this command:

```
create form ORD as hwfrmMaint from HWCTRL62
```

You can see that this command uses the class hwfrmMaint from the library HWCTRL62.

3. The Form Designer appears with the caption ORD in the window title bar.

Again, notice that you have a form with a bunch of command buttons and a combo box. If you open the Properties window, you'll see several new methods in the bottom of the Methods tab: Back(), Next(), and so on. However, upon opening up any of the methods, you won't find any code in them. Why? Because the code is in the class definition—not in the forms created from the class.

Continue creating the rest of the form as in the previous example. Why do you have to do this again, instead of just reusing that form? Because I'm the teacher and if you don't do what I say, you'll stay after school for a month! Well, that and also because this form is using a new base class definition.

4. Change the caption of the form to "Order Maintenance."
5. Create a data environment for the form.
  - Select Data Environment from the Form Designer's shortcut menu or select the View, Data Environment menu option.
  - Select the Add Table menu option from the data environment's shortcut menu or the Data Environment menu pad.
  - Select the CUST table.
6. Add fields to the form. You could do this the hard way, by adding them one at a time and then setting the control source. Or you could do it the easy way—by opening the old form, lassoing all of the controls, and copying them to the new form.
7. Save and run the form. Remember that the command buttons in your Order Maintenance form should not contain any code. Why? Because you're creating this form from the hwfrmMaint class—and that's where the code is! The Click() methods in the command buttons—in the hwfrmMaint class—make calls to Next() and Save() methods that are part of the form class by using commands like "thisform.next()" and "thisform.save()".

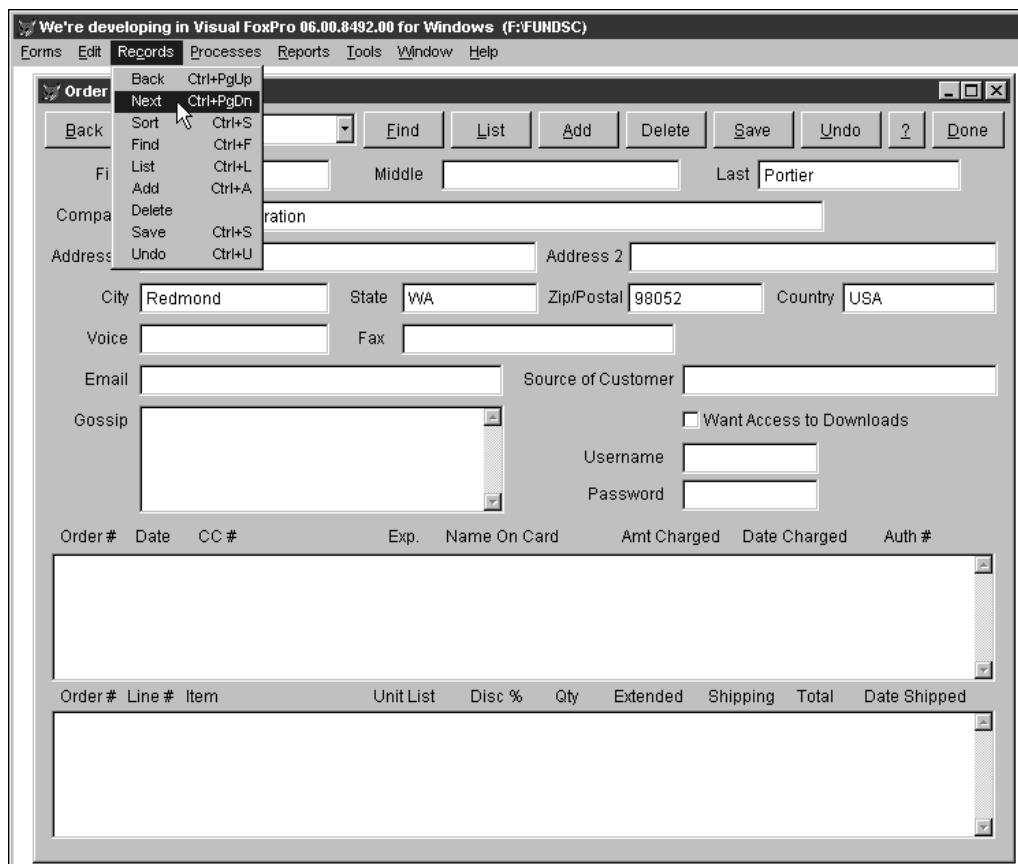
Once you've successfully run the form, you're only half done. Now you'll want to build the project and run the form from the menu. Here's how:

1. Open the Menu Builder and add the following command to the Forms, Orders menu option:

```
do form ORD
```
2. Build the project by clicking the Build button in the Project Manager window. Select the "Rebuild project" option button and make sure the "Recompile all files" check box is checked.
3. Run the application by executing the DO IT command in the Command window. You'll be presented with the Login screen, and after you close that form, the main menu will appear. Select Forms, Orders, and the Order Maintenance form will appear.

But that's not all! Now that you've got your form up and running, you'll see that you can run the form with the command buttons on the form as you would expect. But you can also

execute the same functionality by selecting menu commands from the Records menu, as shown in **Figure 14.4**.



**Figure 14.4.** You can access the navigation methods through the command buttons on the form or via the Records menu.

So try this out. You can open the form and then fire off the Back(), Next(), Save(), or Undo() methods, either from the menu or the command buttons. You can fire the dummy Find(), List(), Add(), or Delete() methods through those command buttons as well. And because there are slightly different methods in each, you can tell which action the user initiated—a menu choice or a button click.

Hey, did you hear what I just said? There are different methods in the Find menu option (“You have selected the Find menu option”) and the Find command button (“You have pressed the Find command button”). This just doesn’t feel right, does it? In fact, it feels downright stupid. This isn’t the ‘80s where excess was considered a virtue.

You could—and should—slim down your code here. Do so by putting `_screen.activeform.next()` in the command button Click events as well. But remember: You

don't open the form and put “`_screen.activeform.next()`” in the Click event of the command button on that form. Those methods should go in the Click event of your `hwfrmMaint` class, right? That'll get taken care of shortly.

Then, when you DO IT, you can issue the Forms, Orders command to open the Order Maintenance form. And then you can fire off Next and Previous—from the menu or from the command buttons. The bottom line here is that we're using different mechanisms—a menu option at one point, and a command button at another—to fire the same method in the form. And the method isn't really in the form; it's in the class from which this form was created. Getting the hang of it yet?

## **Adding a toolbar**

It's time to add a toolbar, which is simply another mechanism from which you'll call the form's `Next()` method. Obviously, you'll create a toolbar, and reference the following command in the toolbar icon Click event:

```
_screen.activeform.next()
```



In this section, I'll show you how to have a toolbar open when a form is first opened, and keep that toolbar hanging around for as long as the form is open. We'll be working with the CH14B directory of the source code for this book.

You can think of a toolbar as a simple form that is available at the same time as the main form (as well as the menu). And just as you create a form class from which you create your forms, you're going to create a toolbar class from which you'll create your toolbars.

Remember that a form is made up of controls like command buttons and page frames, but instead of using Visual FoxPro's base classes of those controls, you have your own class library that contains your base classes of command buttons and pageframes. This class library is called `HWCTRL62.VCX`, and the base classes are called `hwcmd`, `hwpgf`, and so on. When you created `hwfrmMaint`, you used your form's base class, `hwfrm`, as a start, and added some of your own control base classes.

You can think of a toolbar class in the same way as your `hwfrmMaint` class: It's a form made up of two controls (an empty toolbar and a toolbar button), but these controls come from your own base classes, not straight from Visual FoxPro. Obviously, you're going to need to create these new controls to be used for your toolbar base class. You're going to add both of these controls to your `HWCTRL62.VCX` library as their own classes, and then create a toolbar class library using those controls.

You could break out your toolbars into a separate toolbar class library, which would eventually contain several toolbars that you would use for different purposes—much like Visual FoxPro contains multiple toolbars for various purposes. Consider the Form Designer toolbar, the Report Writer toolbar, the Standard toolbar, and so on. For the time being, you're just going to put a Navigation toolbar class in your toolbar class library that affords the user the same abilities as the command buttons in the form and the menu options in the Record menu.

So here's what you'll end up with: `HWCTRL62.VCX` will contain three new classes, `hwtbr` (the empty toolbar) and `hwtbrButton` (a toolbar button). These classes will be used to create a third class, `hwtbrNav` (the navigation toolbar). This class will contain buttons for methods such as `Next()`, `Back()`, and `Save()`. You'll then use the `hwtbrNav` class to create a

toolbar form just as you used hwfrmMaint to create ORD.SCX. Finally, you'll learn how to run the toolbar form at the same time that a regular form is run.

### **Creating new control classes in HWCTRL62**

The first step toward having an integrated toolbar is to create three new classes that belong to HWCTRL62: a toolbar base class and a toolbar button base class are two. The toolbar is the shell—the box around the buttons—and the button is the definition for the actual buttons in the toolbar. You might be wondering what the third class is. I'll bet that you're probably going to want to have spacing between groups of buttons as well, just as you used separator bars in the menu to group certain menu options. You don't have to play fancy games to get this effect; you just place the Separator control between groups of buttons. You might decide not to subclass the Separator control yourself, but for consistency's sake, I'll do it here.

1. Create the toolbar class:

- Issue the CREATE CLASS command or select the File, New, Class menu option.
- The New Class dialog appears. Enter the following information:
  - Class Name: hwtbr
  - Based On: select Toolbar
  - Store In: HWCTRL62.VCX

Remember that "hwtbr" will be a record in the HWCTRL62.VCX table. Using OOP terms, hwtbr is a class contained in the HWCTRL62 class library. (This will be the last time I go through this detail, I promise!) And again, you don't have to include the .VCX extension, but I did it here to make it clear what was getting stored in what.

- The Class Designer appears with an empty toolbar in it. Note that, unlike other controls, you can't resize the toolbar in the Class Designer. This is because toolbars are auto-sized to fit the controls and docking position.
- Make changes to the control as you desire.

Remember, this toolbar will be the one from which all future toolbars are created. Thus, you'll make changes to this control that you want reflected in all of your toolbars. For the time being, you'll change the caption to "hwtbr" and leave the rest alone. The "hwtbr" caption will remind you that you're using your own toolbar class when creating toolbars so that you can avoid making a stupid error such as using Visual FoxPro's base toolbar when you meant to use your own. This trick has saved my bacon numerous times.

- Save the class (with File, Save).

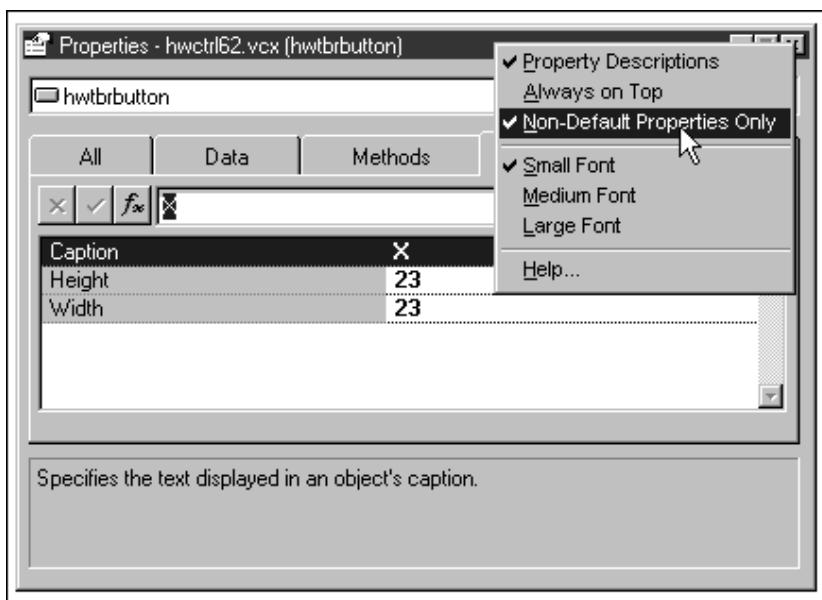
2. Create your toolbar button class:

- Issue the CREATE CLASS command or select the File, New, Class menu option.
- The New Class dialog appears. Enter the following information:

- Class Name: hwtbrButton
- Based On: select Command Button
- Store In: HWCTRL62.VCX
- The Class Designer appears with an empty command button in it.
- Make changes to the control as you desire.

This command button will be used as the template for all toolbar buttons. If you want big buttons, here's the place to make them big. If you want them to be ugly, well, beauty may be skin deep, but ugly goes straight to the class definition. I suggest you change the caption and size of the class as shown in **Figure 14.5**:

- Caption: X
- Height: 23
- Width: 23



**Figure 14.5.** Right-click on the Properties window title bar to choose to display Non-Default Properties only.

You'll notice I used an "X" for the caption instead of "hwtbrButton"—the toolbar buttons are small, so that long caption wouldn't fit well. Is there any code to attach to a toolbar button? Not yet.

By the way, how did I get just the properties I changed to show up in the Properties window? As Figure 14.5 shows, the context menu for the Properties window (displayed by right-clicking on the title bar) allows you to show just Non-Default Properties.

- Save the class (with File, Save). Here you've created yet another record in the HWCTRL62.VCX table.
3. Create your separator class:
- Issue the CREATE CLASS command or select the File, New, Class menu option.
  - The New Class dialog appears. Enter the following information:
    - Class Name: hwsepButton
    - Based On: select Separator
    - Store In: HWCTRL62.VCX
  - The Class Designer appears with a separator in it. Make changes to the control as you desire.

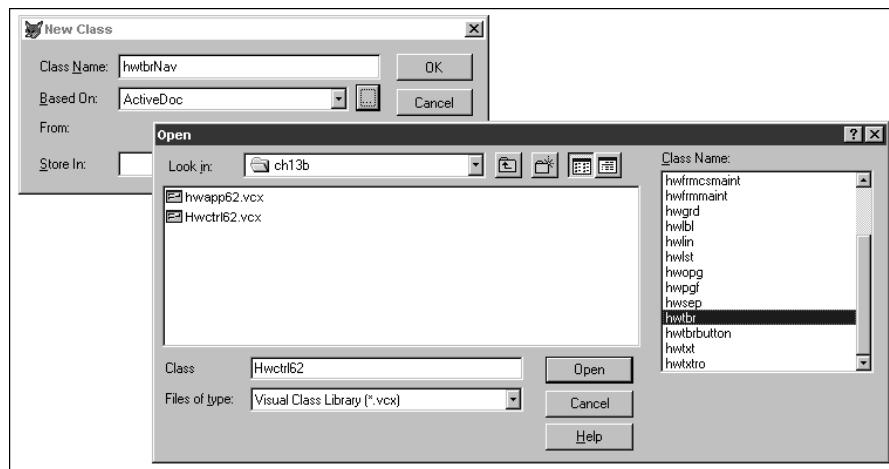
There's not much to change here, really. You might want to cruise the Properties window just to see what a "lightweight" control looks like—no layout properties and virtually no data properties, for example.

And now you've got your three toolbar-related classes. You could also put them into a different library—whatever makes the most sense to you.

### **Creating your base toolbar class**

Now that you've got the components, it's time to create the navigation toolbar itself. Remember that you're going to create a toolbar class—not the actual toolbar—so you're still working in the Class Designer.

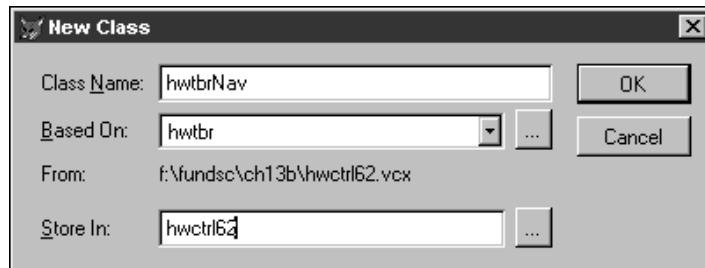
1. Create the toolbar outline:
  - Issue the CREATE CLASS command or select the File, New, Class menu option.
  - The New Class dialog appears. Enter the following information:
    - Class Name: hwtbrNav
    - Based On: click the Based On ellipsis command button and select the hwtbr class from HWCTRL62.VCX, as shown in **Figure 14.6**.



**Figure 14.6.** Select the *hwtbr* class name from the *HWCTRL62.VCX* using the Based On ellipsis button in the New Class dialog.

This step is always a tricky one for new programmers—because we’re creating a class from one of our classes, not from one of the Visual FoxPro base classes that’s listed in the drop-down listbox. Don’t fret if it takes you a few times to remember to display your own classes by using the ellipsis command button—it happens to most people.

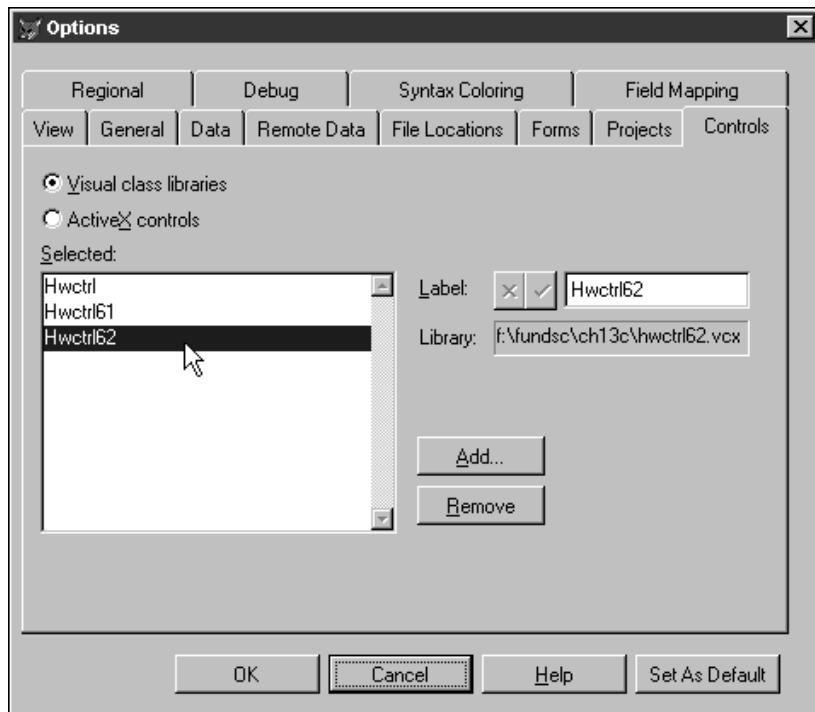
- Store In: HWCTRL62.VCX, as shown in **Figure 14.7**.



**Figure 14.7.** Store the *hwtbrNav* class in the *HWCTRL62* class library.

- The Class Designer appears with an empty toolbar in it. The caption is “*hwtbr*” and serves as a reminder that that this toolbar came from your own *hwtbr* class.
- Change the caption to “Navigation” with the Properties window.

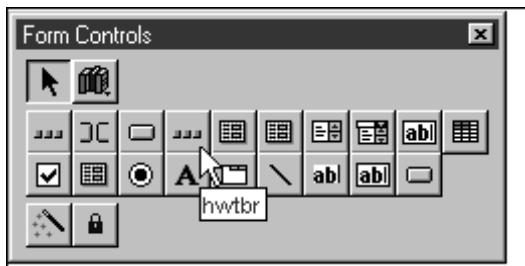
2. Now you're going to add controls to this toolbar, just as we've added controls to a base form in the past. However, the dynamics of a toolbar are a little different than a form, so be prepared to goof around a little bit.
- Open the Form Controls toolbar and click the View Classes button.
  - Select HWCTRL62 from the shortcut menu (if it's not there, you can click Add and select it at this point, but you really should register your HWCTRL62.VCX library via the Controls tab in the Options dialog). If you've been following along by placing the source code for each part of this chapter in a separate directory, you'll also want to be careful that you're working with the correct version of HWCTRL62. Look in the Controls tab of the Options dialog, and highlight the HWCTRL62 item in the listbox, as shown in **Figure 14.8**. Make sure the Library read-only text box shows the path of the correct version—not a library from a previous section.



**Figure 14.8.** Look at the Library read-only text box in the Controls tab to make sure the class library that's registered with Visual FoxPro is the correct one.

You should see a toolbar that looks much like the one in **Figure 14.9**. If you don't see the new "toolbar" and "toolbar button" buttons, you're probably still

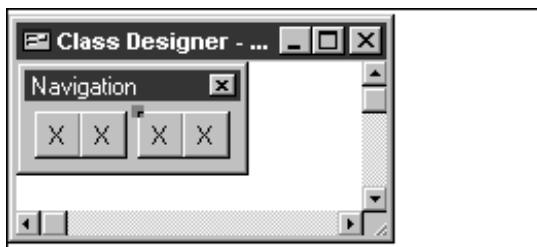
referencing an earlier version. At this point, you're probably thinking that you might want to break out your controls into separate toolbars, or, at the very least, assign different images to the buttons for each of your own classes. It can get pretty confusing with several buttons all carrying the same image.



**Figure 14.9.** The correct version of HWCTRL62 should have two toolbar buttons and a separator button.

- Click (don't drag) the toolbar button icon in the Form Controls toolbar and then click in the toolbar.
3. Add separators between certain toolbar buttons:
- Save the class now, before continuing work. Separators can be tricky at first, and it will be nice to have an intermediate step to come back to if you make a mistake.
  - Click the separator icon. The mouse pointer changes into a cross-hairs icon.
  - Click the button to the right of where you want the separator to be placed. In other words, if you've got five buttons and you want the separator to go between the fourth and fifth buttons, click on the fifth button with the cross-hairs.

After you've added your first separator, you should see something like **Figure 14.10**.



**Figure 14.10.** The gray box between the second and third toolbar buttons is the separator.



*If you put a separator on the toolbar by mistake, you need to select it and then press the Delete key (on the keyboard—remember the keyboard?). It's tricky to select the separator control in the toolbar because it's so thin, so use the Object drop-down list box in the Properties window to select the separator, and then select the Class Designer window. I click on the title bar of the Class Designer window so that I don't accidentally select another control in the toolbar again. You'll see the sizing boxes appear for the separator control. Now you can press the Delete key.*

- Add a total of eight buttons and two separators. You can use the button-lock icon in the Form Controls toolbar to lock the toolbar button, and then just click in your toolbar in the Class Designer eight times.
4. Now it's time to goof around with the toolbar a bit. You'll want a bit of patience and a steady hand. You can select a separator and drag it between two buttons; the easiest way is to use the Object combo box in the Properties window to select the separator control. However, I find it's even easier to drag the buttons back and forth in the toolbar instead of trying to minutely maneuver the tiny separator.
  5. Having a toolbar with eight X's is not necessarily the most friendly user interface. You could change the captions to abbreviations like "N", "P", "F", "L", and so on, but because we're in Windows, let's do what the Windows folk do. You can change the Picture property of a button and attach the name of a graphic file whose image will appear on the button.

If you're playing ahead, you might have already tried to grab an image file (such as a .BMP) through the Picture property in the Properties window. And then you were disappointed when the image didn't display in the toolbar. Here's the trick: You'll also have to get rid of the caption for the button: Select the Caption property in the Properties window, and just press Delete. (Make sure you don't delete the button itself!) You can add your own .BMPs if you like, or use the ones included with this chapter's source code.

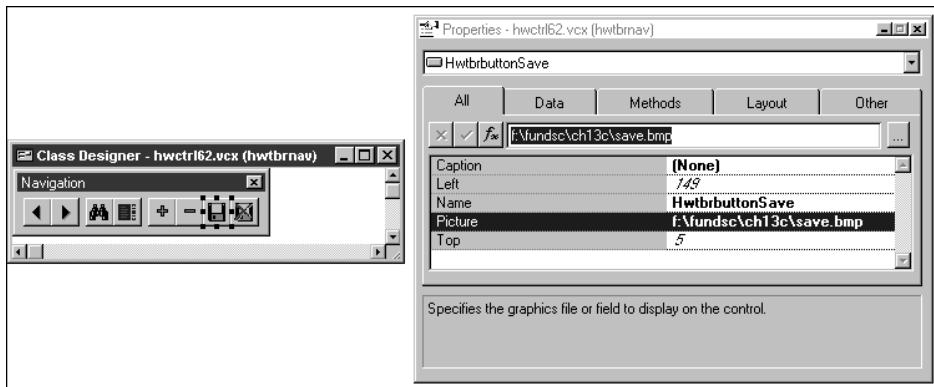
6. Finally, be sure to change the name of each button, as shown in the Properties window in **Figure 14.11**. Change the names to tbrbtnNext, tbrbtnPrevious, and so on. The toolbar will work without doing so, but it will be easier to identify each button when you attach code to them; and when you're activating and deactivating the toolbar, you're going to find it easier to work with code like this:

```
cToolBar.tbrBtnFirst.click()
```

than like this:

```
cToolBar.tbrBtnBase5.click()
```

The end result of your toolbar should look like Figure 14.11.



**Figure 14.11.** Be sure to change the name of each button, and delete the Caption if you're going to use a Picture for the image on the button.

- Now that you've got your Navigation toolbar with nine buttons and three separators, how about making it do something?

- Make the calls to the form's Next(), Previous(), and other methods in the corresponding buttons' Click() event.

```

hwtbuttonNext.click()
screen.activeform.next()

hwtbuttonPrevious.click()
screen.activeform.previous()

hwtbuttonFind.click()
screen.activeform.find()

hwtbuttonList.click()
screen.activeform.list()

hwtbuttonAdd.click()
screen.activeform.add()

hwtbuttonDelete.click()
screen.activeform.delete()

hwtbuttonSave.click()
screen.activeform.save()

hwtbuttonUndo.click()
screen.activeform.undo()

```

- Save the class.

Again, remember that this is a navigational toolbar class—not the toolbar itself. You'll be able to use it as the definition for toolbars, but also subclass it as you

need. For example, you might find one application doesn't want a List toolbar button, or maybe wants an additional toolbar button for Help. You could subclass hwtbrNav, naming the new toolbar hwtbrNavWithHelp, and simply add another button with a question mark on it.

8. Coordinate the calls in the command buttons, menu options, and toolbar buttons.

Remember how you had different methods in the command button Click event and menu option? Here's the time where it all gets pulled together:

- Open your maintenance form base class, hwfrmMaint, by double-clicking on the name in the Project Manager.
- Make sure that you have methods like Next(), Previous(), and so on, that belong to this class, and that you have code in those methods, like so:

```
next()
if !eof()
    skip
    if eof()
        skip -1
    endif
    screen.activeform.refresh()
endif
```

- Change the code in the Click events of the Next command button on frmBaseMaint from this:

```
if !eof()
    skip
    if eof()
        skip -1
    endif
    screen.activeform.refresh()
endif
```

to this:

```
_screen.activeform.next()
```

and do so with each of the other command buttons as well. When you're done, you should have the same call to the form's method in the command button, menu option, and toolbar button.

And now the toolbar is done. It isn't integrated with the form or system yet, but that'll come in a minute.

### Getting your toolbar to work with the app

So now you've got your form, menu, and toolbar all making calls to the `_screen.activeform.next()` method. Looks like you can run your app and you're all set, eh? Except when you do so, the toolbar doesn't show up. The form appears but the toolbar doesn't. Well, of course not—where did you do anything to actually instantiate the toolbar, or make it appear? Hmm? Computers aren't smart, or stupid; they simply do what they're told to do—

but no more and no less. How can you get the toolbar to be available? Well, because it's a form, you could probably include it with your Order Maintenance form and make that a formset. Then you'd run the formset that consists of the Order Maintenance form as well as the toolbar.

Here's how this would work. While it won't be the optimal solution, you'll be able to demonstrate the idea that you're calling the same method from the toolbar icon, the command button, or the menu option.

1. Open the Order Maintenance form by double-clicking on its file name in the Project Manager.
2. Select the Form, Create Formset menu option.
3. In the Forms tab of the Options dialog box, change the form template to hwtbrNav of HWCTRL62.VCX, and click OK.
4. Once you've got the Form Designer window back, select the Form, Add Form menu option. A new form, based on the hwtbrNav toolbar class, is created. (If you can't see it, it may be hidden under Order Maintenance.)
5. Save the form.
6. Run the application and select Forms, Order Maintenance. You'll see the toolbar appear with the form, and you can use the command buttons, toolbar buttons, or menu options to navigate through the table. You can also click the Done command button, the File, Close menu option, or double-click on the form's close box to get rid of the form.

## **Consolidating code in a generic method**

Well, this triple threat of functionality—command button, menu option, or toolbar button—is nice and all, but isn't it really overkill? Why clutter up the form with these command buttons? The whole reason of having a toolbar was to make more room on the forms—and to be able to use the same mechanism across all forms, instead of repeating the same group of command buttons on every form. That's what's next: getting rid of the command buttons. And it's pretty easy—just remove them from the hwfrmMaint class, and you'll be done with them until you change your mind again.

However, while you were playing around, you might have run into another problem. If you ran more than one instance of the Order Maintenance form (or if you were adventurous and created additional forms, such as Invoices and Inventory), you quickly found that a separate toolbar is created for each form, and all of a sudden you've got five or six toolbars floating around on the screen. Functionally, they work; every toolbar works with every form, but it's still pretty messy, wouldn't you say?

Well, I already started you part of the way toward a solution, since you don't have the name of the form hard-coded in the toolbar. Instead, in the Click() method of a particular toolbar button, a call is made to the appropriate method in the form itself—and, yes, it gets better. You don't even have to know which form is open, as long as it has a method with the right name. And because the methods are attached to the form's base class, and you then created the forms from that class, you know you're going to be in good shape. Finally, because

you used the syntax “`_screen.activeform`”, you don’t even have to worry about which form is active, since this is now being handled automatically.

The problem with the multiple toolbars showing up is a result of that decision to make a formset with the form and the toolbar. Bad idea, bad, bad! (Please see Appendix A if you’re still a little fuzzy on this “good-bad” thing.) Instead, what you’d like to do is separate the form and the toolbar, and just call the toolbar once, when the first form is being put up on the screen. After that, you don’t need to instantiate the toolbar again. It sounds like this procedure should be handled by the form, doesn’t it? The first thing to do is add a pair of methods to the `hwfrmMaint` form. (Why not the `hwfrm` form? This is left as an exercise to the reader. Well, okay, maybe not. The reason is that not all forms will have navigation toolbars—in fact, the navigation toolbar is specifically made for forms made from the Maintenance class.)

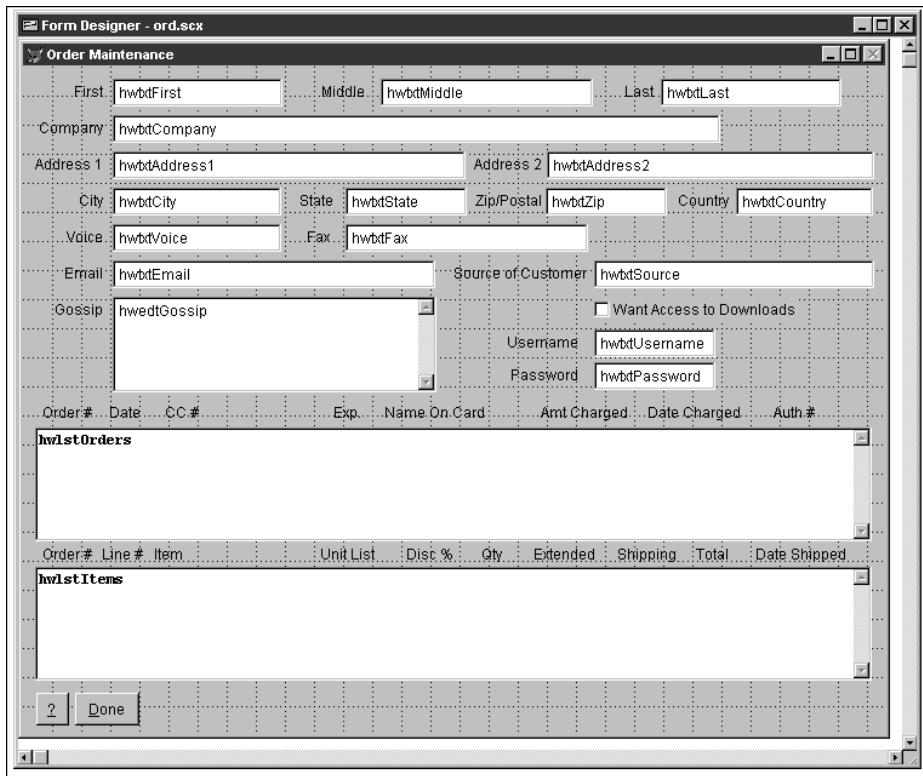
So here’s the “to do” list for this section. First, get rid of the command buttons from your maintenance form base class. Then create an “instance handler” that will count the creation of forms and handle the creation (and destruction) of the toolbar when appropriate. This instance handler will eventually be expanded to control the number of instances of a form that can be opened. Finally, I’ll show you how to get the toolbar to automatically dock at the top of the screen when it’s created.

### **Getting rid of the command buttons**

 As usual, I’m starting with the work I finished up with in the previous section, so the first thing I’ll have you do is copy all of your code into a new directory, or use the code in the CH14C directory of the source code for this book. Be sure to point to the new version of HWCTRL62 if necessary.

1. First, modify your maintenance form base class and get rid of the command buttons in the maintenance form base class:
  - Open the `hwfrmMaint` class by double-clicking its name in the Project Manager.
  - Delete all command buttons at the top of the form. You can delete the Sort combo box as well. Note that you’re not going to be able to delete the Help or Done buttons because they’re part of the `hwfrmMaint` class definition.
  - Save the class.
2. Remove the toolbar form from the Order Maintenance formset:
  - Open the Order Maintenance formset by double-clicking on its name in the Project Manager.
  - Select the Navigation toolbar form. (It might be underneath the Order Maintenance form; you can drag the Order Maintenance form off to the side by its title bar.)
  - Select the Form, Remove Form menu option.
  - Select the Form, Remove Form Set menu option.
  - Save the form.

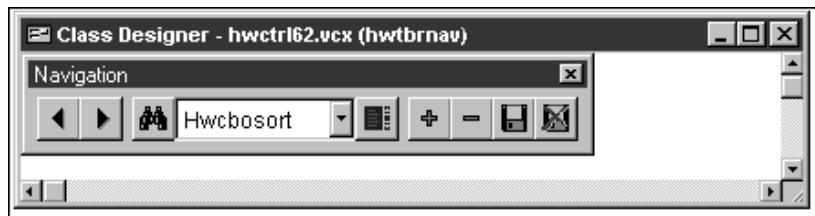
3. Rearrange the controls on the Order Maintenance form as desired. See **Figure 14.12**.



**Figure 14.12.** The maintenance form base class as it should be—without space-hogging command buttons.

You might think the form looks a little stupid with those two dinky command buttons at the bottom. At the current time, you're certainly correct. You might also be wondering why I didn't put them on the toolbar. Another valid question. Admittedly, you could do that. However, this form will be used to launch other processes, and those command buttons will be located on the bottom of the form next to the Help and Done buttons. You're certainly welcome to design your forms as you like.

4. Add the Sort combo box to the Navigation toolbar:
- Open the hwtbrNav class.
  - Drop a copy of your own combo-box class on the toolbar. You might have to futz with it a little bit in order to get the combo box placed in between the buttons you want. I personally put the combo box between the Find and List buttons. See **Figure 14.13**.



**Figure 14.13.** Place a copy of your combo-box class on the hwtbrNav toolbar.

- Adjust the height of the toolbar buttons to match the height of the combo box.

If you look carefully at your screen, you'll notice that the combo box is slightly larger than the buttons. The default height for the combo-box control is 24 pixels, but I suggested a height (and width) of 23 pixels for the toolbar buttons. You might be thinking that you're glad that the toolbar is a class, because you only have to change the height of each of these buttons in the hwtbrNav class? Ha! Life is even better than that. You actually only have to change the height of the button in the toolbar button class—the height will then be inherited by every button on every toolbar.

### Creating an instance handler

I'm going to start out with an instance handler that's modeled after the one used in the Tastrade sample app. However, because this architecture is more compartmentalized than Tastrade, the places that your properties and methods will be located are going to be substantially different than Tastrade.

This instance handler isn't a separate entity in the application. Rather, it's just a few properties and methods. You'll add a couple of properties to the application base class that will track how many instances of forms are running, and which forms are running. You'll also add methods to handle the incrementing and decrementing of these counters, and to actually display and hide the toolbar when it's appropriate.

You might be saying right now, "Whoa! 'Application base class?' Whuzzat again?" The application base class is kinda sorta the replacement for your procedure libraries, in that it is a repository for application-wide data and functions. It's instantiated in IT.PRG, your startup program, and thus is available for the rest of the application. Here's how to create it:

1. Add the nFormInstanceCount and oToolBar properties to the App class of HWAPP62:
  - Open the App class by double-clicking it in the Project Manager, entering the command MODIFY CLASS in the Command window, or selecting the File, Open menu option.
  - Select the New Property menu option from the Class menu pad.
  - Enter nFormInstanceCount as the Name.

- Enter a description and save the new property by clicking the OK button.
- Repeat the last three steps for oToolBar.
- Initialize nFormInstanceCount to 0 and oToolBar to .NULL. in the Properties window.

These are application-specific properties as opposed to form-specific because they will apply to the whole application—not just a specific form.

2. Add the ShowAppToolBar() and RemoveAppToolBar() methods to the App class of HWAPP62.

Once the method has been created, use the New Method menu option from the Class menu pad and the Code window to create the following methods:

```
ShowAppBarToolBar()
lparameters tcToolBar

*
* this is called when we open a form
* if there are no forms already open,
* then we want to show the toolbar for this form
*
if this.nFormInstanceCount = 0
  set sysmenu on
  this.oToolBar = createobject(tcToolBar)
  this.oToolBar.Show()
  activate menu _msysmenu nowait
  set sysmenu automatic
endif
this.nFormInstanceCount = this.nFormInstanceCount + 1

RemoveAppBarToolBar()
*
* this is called when we close a form
* if this is the last form available, rid the toolbar
*

this.nFormInstanceCount = this.nFormInstanceCount - 1
if this.nFormInstanceCount = 0
  this.oToolBar = .NULL.
endif
```

So now you've got procedures for handling your instances. The ShowAppToolBar() method displays the toolbar if this is the first form being run, and the RemoveAppToolBar() method gets rid of the toolbar if it's the last form to go away.

But having these methods is one thing—the thing that you're going to find over and over is that the hardest part is figuring out where the methods are called. In other words, these methods don't exist in a vacuum, nor do the application's base class properties like aInstance and nFormInstanceCount. But where are these methods run from?

I could just tell you, but that wouldn't be any fun. Let's try to puzzle this out together. By learning the logic behind the placement of the methods, you'll have an edge when it comes time to look at other Visual FoxPro applications with which you're not familiar—or even your own application after you've been away on vacation for a couple of weeks.

How about that ShowAppToolBar() method? Its purpose is to display the toolbar when a form is loaded—if there are no forms already open. When should this be called? Well, you just said the answer out loud: when a form is loaded.

3. Modify the base form's (hwfrmMaint) Init() and Destroy() methods to handle the toolbar:

- Change the Init() method to this:

```
if !empty( thisform.cToolBar )
    oApp.ShowAppToolBar( thisform.cToolBar )
endif
```

- Change the Destroy() method to this:

```
if !empty( thisform.cToolBar )
    oApp.RemoveAppToolBar()
endif
```

4. Add a cToolbar property.
5. Initialize the form class's cToolBar property with the name of the toolbar that goes along with the form.

You'll notice that your base maintenance form has a property, cToolbar, and expects a value in that property—specifically, the name of the toolbar to be created. The logical place to put a value in this property is in the form class, because all of the forms created from that form class will probably have the same toolbar. However, if you initialize the property in the form class, you can still override it in both subclasses of the form class (for specialized maintenance forms) as well as specific instances of the form.

To do this here, simply open the Properties window of the class and set the cToolbar property to "hwtbrnav" (without the quotes).

6. Save the class.

### Getting the toolbar to dock automatically

It would be a nice touch to have the toolbar automatically dock on the top of the screen when it's opened. This can be accomplished simply by calling the Dock() method with a parameter of 0. You'd expect to do this in the toolbar class definition, right?

1. Open the hwtbrNav toolbar class by double-clicking its name in the Project Manager.
2. Open the Init() method in the Code window.
3. Enter the following code in the Init() method:  
`this.dock(0)`
4. Save the toolbar class.

### Adding the form name to the Window menu

You know what else would be nice? Having the name of the form that was just opened also show up in the Window menu. If you open multiple forms, the names of each of those would show up in the Window menu—just like when you have multiple documents open in Word or multiple windows open in Visual FoxPro. Because your forms are likely to be pretty big, you're going to more than likely need this to toggle through them.

Here's how to do it. Because this functionality is specific to the opening and closing of a form, you're going to add a pair of methods to your baseform class, hwfrm. The first method will handle adding the form name to the menu, and the second will handle removing the form name from the menu when the form is closed.

1. Modify the class hwfrm by double-clicking its name in the Project Manager. (You are using the Project Manager, aren't you?)
2. Add the following methods to the class by selecting the Add Method menu option in the Class menu and then typing code in the Code window:

```
AddFormNameToMenu()
Local lnNoMenu, lcNameForm
if cntbar("Window") = 0 or getbar("Window", cntbar("Window")) < 0
  nNoMenu = cntbar("Window") + 1
else
  nNoMenu = getbar("Window", cntbar("Window")) + 1
endif
define bar nNoMenu of Window prompt thisform.caption ;
  after _MLAST
cNameForm = thisform.Name
on selection bar nNoMenu of Window activate window &cNameForm

RemoveFormNameFromMenu()
local lnBar
for nBar = cntbar("Window") to 1 step -1
  if prmbar("Window", getbar("Window", nBar)) = thisform.caption
    release bar getbar("Window", nBar) of Window
    exit
  endif
endfor
```

3. Modify the following methods in the hwfrmMaint's Init(), Destroy(), and Activate() methods to handle both the window and the toolbar:

- Change the Init() method to this:

```
thisform.AddFormNameToMenu()  
oApp.ShowAppToolBar( thisform.cToolBar )
```

- Change the Destroy() method to this:

```
thisform.RemoveFormNameToMenu()  
oApp.RemoveAppToolBar()
```

- Change the Activate() method to this:

```
activate menu _msysmenu nowait
```

- Save the class.

What is going here is fairly straightforward, although the syntax is a pain because it's not often used. In the AddFormNameToMenu() method, I first determine how many bars already belong to the Window menu, and increment that number by one. Then I add the current form's name as another menu bar, and, finally, tell Visual FoxPro to activate that form if the user selects the menu option.

In the RemoveFormNameFromMenu() method, I do the opposite: decrement the number of menu bars under the Window menu and then remove the menu bar for the form as well.

In the Init() method of the form, I actually run the AddFormNameToMenu() method, just as when the toolbar was displayed.

Finally, when the form is closed, you're going to want to get rid of the form's name in the Window menu, and that's when the RemoveFormNameFromMenu() method is called.

There are a couple of important tips to point out. First, if your Window menu pad is called something other than "Window," you'll want to use that name instead of "Window" in the code above. Second, if you use the standard "\_msm\_Windo" Window menu from the Visual FoxPro menu, this functionality comes along for the ride for free! In fact, if you include these two functions in your forms when you are also using "\_msm\_Windo", you'll end up with two menu commands for each open form in the Window menu. The code for this section demonstrates this—you can comment out the AddFormNameToMenu() and RemoveFormNameFromMenu() functions to see what I mean.

## Enhancing your form class with generic methods

This chapter has been about creating a framework upon which you can hang your data-entry forms. It's covered a lot, and if you're feeling overwhelmed, don't worry. Of course, I'm not done yet. I've left hooks or starting points for a number of features that have been left unfinished (or even unstarted) in the interest of not getting sidetracked.

Now it's time to take care of some of those things. You've got a calling program, a main menu, a foundation for handling toolbars and window menus, and some classes to handle standard navigation and maintenance functions. All that's left is to add a few forms and a report or two, and you can ship it!

I'm reminded of the cartoon where a scientist is showing off a mathematical proof on the blackboard to a number of colleagues. There are a host of complex equations all over the board, but between steps 5 and 7, a small box reads, "Step 6: A miracle occurs." And one of his colleagues says, "I think you need more details in Step 6."

Here are the details...

## An introduction to MessageBox() and #INCLUDE

I'm going to be using the MessageBox() function heavily in the next few sections, so it's worthwhile to explain some of the details here. I'm synopsizing here, so you might want to take a minute and check it out in the Language Reference.

The first parameter is a custom message of your own choosing that will display in the message box. The second, optional parameter indicates what types of actions the user will be able to perform. This parameter is a numeric value, and various values can be added together so the message box can allow multiple behaviors.

The values 0 through 5 in the second parameter determine the buttons that appear in the message dialog box. For example, if there is no second parameter or if it has a value of 0, just an OK button will appear. A value of 4 will cause Yes and No buttons to appear.

The values 16 through 64 (in increments of 16) determine which icon will appear. A value of 16 will cause a stop sign to appear, a value of 32 corresponds to a question mark, and so on. All of these values are described in Visual FoxPro Help under MessageBox().

The values 0, 256, and 512 correspond to which button is the default in the dialog. 0 (or no parameter) means the first button is the default, 256 corresponds to the second button, and 512 corresponds to the third.

If you've been watching carefully, you'll notice that none of these values overlap. This means that these values can all be added to create a unique characteristic for the message box. For example, a second parameter of 276 can only be created one way: 4 (Yes/No buttons) + 16 (stop sign icon) + 256 (second button as default).

It would be a good exercise for you to memorize every value that can be passed as a second parameter. You can never tell when a stranger is going to stop you as you're driving by and ask what the value 537 corresponds to. However, if you're not so inclined, you might want to use a shortcut.

The file FOXPRO.H includes definitions of a number of constants as English words. Thus, the constant MB\_OK ("MB" stands for "message box" or "message button," if you like) is assigned the value 0, and as long as you include the file as part of your application, you can then refer to English constants instead of having to remember the numeric values.

You can include the FOXPRO.H file in your app in several different ways. You can use the following line in each program that uses these constants:

```
#INCLUDE FOXPRO.H
```

You can add an Include file to a specific form with the Form, Include File menu command, or automatically add it to every form and class by specifying a Default Header File in the Default File Locations tab of the Tools, Options dialog.

You can find FOXPRO.H in the main Visual FoxPro directory, and I encourage you to open it up and look around to see what has already been defined. Here are a few of the constants I find handy:

### Toolbar positions

Position	Setting	Description
#DEFINE TOOL_NOTDOCKED	-1	Not docked
#DEFINE TOOL_TOP	0	Docked on top
#DEFINE TOOL_LEFT	1	Docked to the left
#DEFINE TOOL_RIGHT	2	Docked to the right
#DEFINE TOOL_BOTTOM	3	Docked on bottom

### MessageBox parameters

Parameters	Setting	Description
#DEFINE MB_OK	0	OK button only
#DEFINE MB_OKCANCEL	1	OK and Cancel buttons
#DEFINE MB_ABORTRETRYIGNORE	2	Abort, Retry, and Ignore buttons
#DEFINE MB_YESNOCANCEL	3	Yes, No, and Cancel buttons
#DEFINE MB_YESNO	4	Yes and No buttons
#DEFINE MB_RETRYCANCEL	5	Retry and Cancel buttons
#DEFINE MB_ICONSTOP	16	Critical message
#DEFINE MB_ICONQUESTION	32	Warning query
#DEFINE MB_ICONEXCLAMATION	48	Warning message
#DEFINE MB_ICONINFORMATION	64	Information message
#DEFINE MB_APPLMODAL	0	Application modal message box
#DEFINE MB_DEFBUTTON1	0	First button is default
#DEFINE MB_DEFBUTTON2	256	Second button is default
#DEFINE MB_DEFBUTTON3	512	Third button is default
#DEFINE MB_SYSTEMMODAL	4096	System Modal

### MsgBox return values

Value	Setting	Description
#DEFINE IDOK	1	OK button clicked
#DEFINE IDCANCEL	2	Cancel button clicked
#DEFINE IDABORT	3	Abort button clicked
#DEFINE IDRETRY	4	Retry button clicked
#DEFINE IDIGNORE	5	Ignore button clicked
#DEFINE IDYES	6	Yes button clicked
#DEFINE IDNO	7	No button clicked

## Cursor buffering modes

Mode	Setting	Description
DEFINE DB_BUFOFF	1	No buffering
DEFINE DB_BUFLOCKRECORD	2	Pessimistic record buffering
DEFINE DB_BUFOPTRECORD	3	Optimistic record buffering
DEFINE DB_BUFLOCKTABLE	4	Pessimistic table buffering
DEFINE DB_BUFOPTTABLE	5	Optimistic table buffering

By the way, you can create your own constant definitions as well. For example:

```
#DEFINE HOURS_IN_A_DAY 24
```

However, you probably don't want to add your own constants to the FOXPRO.H file. Instead, you can create your own Include file (it's just a text file), and then use the command:

```
#INCLUDE YOURSTUF.H
```

if the text file that contains your constants is named YOURSTUF.H.

You can't specify more than one Include file at a time, so if you want to include constants from both FOXPRO.H and YOURSTUF.H, include this line in YOURSTUF.H in order to incorporate the FOXPRO.H constants in YOURSTUF.H:

```
#INCLUDE FOXPRO.H
```

## Data handling—buffering and multi-user issues

 The way Visual FoxPro handles the movement of data between forms and tables is different than in FoxPro for Windows or DOS. Up to this point, I was rather blasé about data handling—I just did a TABLEUPDATE() or TABLEREVERT(), and hoped I got lucky. Well, one of these days, we're not going to be so lucky. What if the TABLEUPDATE() or TABLEREVERT() fails? The code used in this section can be found in the CH14 source code downloads for this book.

The first option is to do things “the old way”—in other words, to create memory variables for each field on the form, and then allow the user to edit against those variables. You would either lock the record when the user started to edit, or lock it when the user attempted to save, and handle conflicts with other users at that point. Well, that's actually a lot more work in Visual FoxPro because of the way that controls work now. Furthermore, Visual FoxPro provides a number of tools that enable a more robust strategy.

Visual FoxPro allows you to edit against a buffer that it automatically sets up. When the user goes to save, Visual FoxPro takes care of locking the record and moving data back to the table. The only thing you have to worry about is error messages that Visual FoxPro generates when conflicts occur. Unlike earlier versions, you don't have to trap for the conflicts yourself.

You have four choices as far as buffering goes. These are the four combinations of optimistic and pessimistic locking and row and table buffering:

- Pessimistic row buffering
- Optimistic row buffering
- Pessimistic table buffering
- Optimistic table buffering

I'm going to concentrate on row buffering to start, because the sample form works on just one record at a time, and then I'll discuss the differences with table buffering as they pop up. Table buffering works similarly except that you need to scroll through all of the possible records involved in changes, instead of just paying attention to one.

### A simple error handler

Visual FoxPro handles all of the data transfer between the buffer and the table. If user actions create contention (say, a user tries to edit a locked record), Visual FoxPro will handle the situation and generate the appropriate error. All that's left to do is provide a friendly interface for the user so they understand what happened, and then let them decide what action they want to take.

I'll demonstrate this capability by modifying the dummy error handler that was called in the instantiation of the ENV class during program startup. The following code was in the DoSets() method:

```
if oApp.cPermLevel = "ADMIN"
  on error do DevError with lineno()
  set escape on
else
  on error do UserError with lineno()
  set escape off
  on escape *
endif
```

The following procedures are in MYPROC.PRG, which is opened when the application is started, so they're available throughout the application. (If you're wondering, yes, these really don't belong here anymore—I'll discuss converting MYPROC to a class library near the end of this chapter.)

```
procedure DevError
lparameters llineno
local array laError[1]
=aerror( laError )
messagebox("Something bad happened." ;
  + chr(13) + " in line " + str(llineno) ;
  + chr(13) + " of " + sys(16) ;
  + chr(13) + " that said " + message(1) ;
  + chr(13) + " because " + laError[2] ;
  + chr(13) + " and the secret code is " + str(error()) , ;
  0,"Developer Warning!")
return .t.
```

```
procedure UserError
lparameters llineno
local array laError[1]
=aerror( laError )
messagebox("This system needs improvement" ;
+ chr(13) + " in line " + str(llineno) ;
+ chr(13) + " of " + sys(16) ;
+ chr(13) + " that said " + message(1) ;
+ chr(13) + " because " + laError[2] ;
+ chr(13) + " and the secret code is " + str(error()), ;
0,"Helpful User Message Warning!")
return .t.
```

The first two commands create an array that contains information about the most recent Visual FoxPro error. The array contains seven columns and one or more rows. The type of error that occurred determines the number of rows in the array. AERROR() returns a numeric value that is the number of rows in the array. The contents of the columns vary according to the type of error, but for all errors, the following holds:

Column #	Contents
1	The number of the error (same as the value of ERROR()).
2	The text of the error message.

The message box line then displays the standard message-box dialog with the text of the error message and an OK command button.



*Once you're comfortable with this technique, you may want to investigate a more robust, but necessarily more complex, error handler. The classic example is described in a white paper by Doug Hennig that you can download at his Web site, [www.stonefield.com](http://www.stonefield.com).*

### Causing a conflict—pessimistic row buffering

Let's take a look at the possibilities of pessimistic row buffering first. If you're just getting your feet wet, you might find yourself leaning toward this method because it's safe and easy. Since Visual FoxPro will lock the record as soon as the user starts editing, our error handler takes care of everything nicely. The second user will get a message indicating that they can't access the record, and they'll have to try again later.

Specifically, when a user attempts to edit a record that is locked, Visual FoxPro will generate an error saying "Record is in use by another." You can make this happen yourself. First, open the Order Maintenance form and make sure that the BufferMode property of the form is set to 1-Pessimistic and the Data Session property is set to 2-Private Data Session. Then run two copies of Visual FoxPro, open the same form in both sessions, edit (but don't save or move off the edited record) a record in one session, and then try to edit the same record in the other session. (You can also open the form twice in the same session of VFP, but I think using two copies of Fox is less prone to error.) The result should look like Figure 14.14.



**Figure 14.14.** The result of two users trying to edit the same record when pessimistic buffering is in effect.

Of course, things that are easy usually have their problems, and the problem with this approach is twofold. First, there's the old "out to lunch" syndrome. What if a user starts editing a record but leaves before saving it? As long as he's in the edit process, no one else can access that record. This is a serious problem if the record has a lot of activity, or if the person tends toward long breaks and lunches. The second problem involves large records, in which different segments are edited by different people. For example, the name and address information in a customer record might be edited by the mail room, the phone information could be edited by the telecommunications department, and the credit information would be handled by the finance department. In a pessimistic locking scenario, once the mail room has its hands on a record, the telecommunications and finance departments are out of luck.

How about optimistic row buffering?

### Optimistic row buffering

Optimistic row buffering lets everyone and their brother edit a record, and the person fastest to the Save key is the winner. Visual FoxPro is particularly intelligent in this respect, because it will detect when a second user is trying to save changes over the changes of the first person, and warn the second user that there is a conflict.

However, a message like "Update Conflict," as shown in **Figure 14.15**, might not be the type of message you want your user to see.



**Figure 14.15.** This message occurs when a second user tries to save changes over changes that a first user made with optimistic row buffering.

So look at the ways this situation can be handled. Here's what's happened: A field in the table originally contains "A", and then users One and Two have at it. User One changes "A" to "B" and saves it. User Two, being somewhat of a slowpoke, enters "C" and attempts to save it after User One is finished.

Visual FoxPro knows that the original value was "A" and that the current value is now "B", but it also knows that User Two still thinks that he is overwriting "A" with his change ("C") and doesn't realize that the table now contains "B." This is a potential problem because "B" might be better data than "C."

At this point, Visual FoxPro takes off the gloves and provides User Two with that oh-so-helpful message.

### A generic Save() method

How about intercepting the message with a friendlier message such as "Another user has changed this record while you were editing it. Overwrite those changes with yours?" Then if the user selects "Yes," issue the TABLEUPDATE command with the lForce parameter. This is what the code would look like in the Save() method:

```
m.lcMessage = "Another user has changed this record while you were editing it.  
Overwrite those changes with yours?"  
if !tableupdate()  
    if messagebox( m.lcMessage, MB_YESNO ) = IDYES  
        =tableupdate(.t., .t.)  
        wait window nowait "Changes saved"  
    else  
        =tablerevert()  
        wait window nowait "Changes abandoned"  
        thisform.refresh()  
    endif  
else  
    wait window nowait "Changes saved"  
endif
```

Let me explain what's happening here. First, I initialized a text message that I'll use in the MessageBox() function. Sure, I could have just placed the text in the MessageBox() function itself, but that often gets hard to read.

Next, I try to do a TABLEUPDATE(). This isn't immediately obvious. The command

```
if ! tableupdate()
```

actually issues the TABLEUPDATE() function and gets the return value. If TABLEUPDATE() is successful, it returns .T. And skips the first part of the IF. If it fails, it will process the first part of the IF. If it makes you more comfortable, you could make this a little easier to read, like so:

```
m.lUpdateWorked = tableupdate()  
if m.lUpdateWorked  
    *  
    * do the code if the update worked  
    *  
else  
    *
```

---

```
* do the code if the update did NOT work
*
endif
```

The first half of the IF just confirms to the user that the Save() was successful. It's the second half that's tricky.

The MessageBox line tells the user that he was unlucky in saving, but that he had an opportunity to overwrite the previous user's changes. The MessageBox() function returns a value according to the action he selected. If the value is 6 (the constant IDYES in FOXPRO.H), then we use the TABLEUPDATE() function with two .T. parameters. The first is used when we're dealing with table buffering—it forces TABLEUPDATE() to update all rows—and is irrelevant with row buffering. The second is the lForce parameter that forces Visual FoxPro to overwrite changes made by any other user.

If the user elects not to overwrite the other user's changes, I'll use the TABLEREVERT() function to change back to the current values in the table, and then indicate to the user that his changes have been abandoned. The form will also have to be refreshed, as you might expect.

This mechanism works nicely until a user comes around one day and whines, "I'd like to know what the other user changed..." Kinda makes you long to be a mainframe programmer, where you could smile sweetly and say, "Sure, I'll put it in the queue and we should get to it in about 10 years," doesn't it? Well, actually, it's not that bad.

### **Telling the user which fields have changed**

The OLDVAL() and CURVAL() functions describe the current and old values of a field. When the users start, fieldX in the table contains "A." User One changes fieldX from "A" to "B" and User Two changes fieldX from "A" to "C." User Two saves first. When User One tries to save, OLDVAL("fieldX") contains "A" and CURVAL("fieldX") contains "C." Thus, when User One tries to save, OLDVAL and CURVAL can be compared to see if they're different for fieldX. If so, you can grab the name of the field and stuff it into a string that can be made part of the error message the user sees. This way, if more than one field has changed, you can tell users which fields have changed, and let them make their decision based on that information.

Here's what the modified Save() routine would look like:

```
m.lSaveIsGood = tableupdate()

if m.lSaveIsGood
  wait window nowait "Changes saved"
else
  *
  * the save failed, so we need to tell the user that
  * the data in the table has been changed since they
  * got their copy - and then we give them the option to
  * overwrite or ignore their changes
  *
  m.cStrToDisplay = ""
  for I = 1 to fcount()

  *
  * build a string that consists of all of the
  * fields that have been changed by the other user
  *
```

```
* loop through every field
*
* compare the current value on the disk with
* the old value on the disk
*
* if another user saved changes,
* - curval is their newly saved value
* - oldval is the value before they saved - the value
* that this user is looking at
*
m.cNaField = field(i)
if curval( m.cNaField ) <> oldval( m.cNaField )
*
* if this field has been changed, concatenate
* the name of this field to the existing string
* that will be displayed in the Message Box
*
* add a comma if the string was not empty to begin with
*
m.cStrToDisplay = m.cStrToDisplay ;
+ iif(empt(m.cStrToDisplay), "", ", " ) ;
+ m.cNaField
endif
endfor

*
* if StrToDisplay is not empty, there are differences
* and we have to give the user the opportunity to decide
* what to do
*
if !empty( m.cStrToDisplay )
  m.cMess = "These fields have been changed by another" ;
  + "user: " ;
  + m.cStrToDisplay ;
  + "Do you want to save your changes " ;
  + " and overwrite the data in these fields? "

if messagebox( m.cMess,
  MB_YESNO + MB_ICONEXCLAMATION + MB_DEFBUTTON2 ) ;
  = IDYES
*
* if they want to overwrite, use tableupdate to force
*
=tableupdate(.t., .t.)
wait window nowait "Changes saved"
else
  =tablerevert()
  thisform.refresh()
  wait window nowait "Changes abandoned"
endif && messagebox
else
*
* there were no differences so we will just
* go ahead and force the save
*
=tableupdate(.t., .t.)
wait window nowait "Changes saved"

endif !empty( m.cStrToDisplay)
```

---

```
endif && m.lSaveIsGood
```

This is pretty straightforward. If the TABLEUPDATE() isn't successful, I run a looping process to see what has changed. The FOR/ENDFOR routine loops through every field and determines whether OLDVAL and CURVAL for that field are different. If so, the name of that field is added to a string. Once the looping is done, I check to see if the string has anything in it. If it does, I inform the user which fields have been changed and let them decide. The rest works like before, either forcing changes or reverting to the old data.

### **Telling the user which fields have changed—and what the values are**

That method works, and puts a spring in our step for a little while—until the user comes back again and complains that he'd like to know the values in those fields. Some people are never satisfied! Well, a modification to the routine above will let you save the values of OLDVAL and CURVAL as well as the field name, and with this, the user will have enough information to determine if he can save his changes on top of the existing changes:

```
#INCLUDE "FOXPRO.H"

m.lSaveIsGood = tableupdate()

if m.lSaveIsGood
  wait window nowait "Changes saved"
else
  *
  * the save failed, so we need to tell the user that
  * the data in the table has been changed since they
  * got their copy - and then we give them the option to
  * overwrite or ignore their changes
  *
  m.cStrToDisplay = ""
  for I = 1 to fcount()
    *
    * build a string that consists of all of the
    * fields that have been changed by the other user
    *
    * loop through every field
    *
    * compare the current value on the disk with
    * the old value on the disk
    *
    * if another user saved changes,
    * - curval is their newly saved value
    * - oldval is the value before they saved - the value
    * that this user is looking at
    *
    m.cNaField = field(i)
    if curval( m.cNaField ) <> oldval( m.cNaField )

    *
    * examine whether or not current user also changed
    * the field, and if not (=1), update the current
    * user's value with the new value that has been saved
    *
    * note that we're assuming that the other user that
```

```
* made the changes has more recent data, since they
* saw the value that this user is looking at and
* still decided to change it
*
if getfldstate( m.cNaField ) = 1

    replace &cNaField with curval( m.cNaField )
else
    m.cStrToDisplay = "&cNaField has been changed by" ;
        + " another user. " + chr(10)
do case
case type( m.cNaField ) = "CM"
    m.cStrToDisplay = m.cStrToDisplay ;
        + "Original Value: " + allt(oldval( m.cNaField )) ;
        + chr(10) ;
        + "Your New Value: " + allt(eval( m.cNaField )) ;
        + chr(10) ;
        + "Other User's Value: " + allt(curval( m.cNaField ))
case type( m.cNaField ) = "NF"
    m.cStrToDisplay = m.cStrToDisplay ;
        + "Original Value: " + ltrim(str(oldval( m.cNaField ),20,2)) + chr(10) ;
        + "Your New Value: " + ltrim(str(eval( m.cNaField ),20,2)) + chr(10) ;
        + "Other User's Value: " + ltrim(str(curval( m.cNaField ),20,2))
case type( m.cNaField ) = "YB"
    m.cStrToDisplay = m.cStrToDisplay ;
        + "Original Value: " + ltrim(str(oldval( m.cNaField ),20,4)) + chr(10) ;
        + "Your New Value: " + ltrim(str(eval( m.cNaField ),20,4)) + chr(10) ;
        + "Other User's Value: " + ltrim(str(curval( m.cNaField ),20,4))
case type( m.cNaField ) = "D"
    m.cStrToDisplay = m.cStrToDisplay ;
        + "Original Value: " + dtoc(oldval( m.cNaField )) + chr(10) ;
        + "Your New Value: " + dtoc(eval( m.cNaField )) + chr(10) ;
        + "Other User's Value: " + dtoc(curval( m.cNaField ))
case type( m.cNaField ) = "T"
    m.cStrToDisplay = m.cStrToDisplay ;
        + "Original Value: " + ttoc(oldval( m.cNaField )) + chr(10) ;
        + "Your New Value: " + ttoc(eval( m.cNaField )) + chr(10) ;
        + "Other User's Value: " + ttoc(curval( m.cNaField ))
endcase
=messagebox( m.cStrToDisplay, MB_OK + MB_ICONINFORMATION )
endif && getfld...
endif && curval <> oldval
endfor && I = 1 to fcount()

*
* if StrToDisplay is not empty, there are differences
* and we have to give the user the opportunity to decide
* what to do
*
if !empty( m.cStrToDisplay )
    if messagebox("Do you want to overwrite the other user's changes with your
changes?", ;
        MB_YESNO + MB_ICONEXCLAMATION + MB_DEFBUTTON2 ) = IDYES
    *
    * if they want to overwrite, use tableupdate to force
    *
    =tableupdate(.t., .t.)
    wait window nowait "Changes saved"
else
    =tablerevert()
```

```
thisform.refresh()
wait window nowait "Changes abandoned"
endif && messagebox
else
/*
* there were no differences so we will just
* go ahead and force the save
*
=tableupdate(.t., .t.)
wait window nowait "Changes saved"

endif !empty( m.cStrToDisplay)

endif && m.lSaveIsGood
```

This routine will inform the user with a message box for each field that has changed, and then ask the user whether he wants to save his changes now that he knows the values that the other user has saved. The clever thing about this routine (and thanks to my friend Robert Green for the original idea) is that we're only saving changes to fields that the current user has changed. Come again, you say?

Suppose that fieldX contains 100 and fieldY contains 200. User One changes fieldX to 150 and User Two changes fieldY to 225. Then User Two saves. FieldX contains 100 and fieldY contains 225. But User One's buffer contains 150 and 200. You don't want to write 200 over the 225 of User Two—you'd like to grab that new value of 225 but still save 150 over 100.

You know that the field has been changed because OLDVAL and CURVAL are different. The GETFLDSTATE() function determines whether the current user changed the field. If so, you need to tell the user that there is a conflict in this field. This information gets stuffed into the cStrToDisplay variable. However, if the other user changed this field, not the current user, you want to grab the value that the other user saved, and that's done with the REPLACE command.

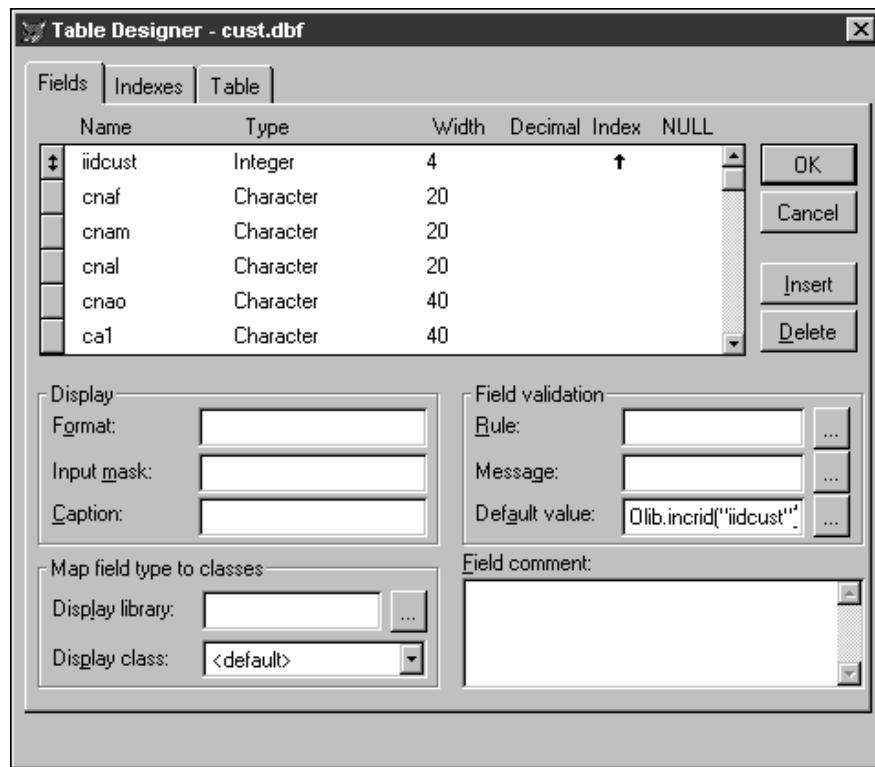
This seems like a lot of code for a simple function, but it executes quite quickly, and notice that it's all completely generic, so you can place it in the Save() method of your frmMaint class and never worry about it again. And it's still a lot easier than manipulating multiple arrays, which was required in earlier versions of FoxPro.

### A generic Add() method

Okay, so that's the Save() method. What about Add()? This one's got to be custom, because you're going to be creating a brand new record and probably stuffing it with custom values each time, right? Not so fast. You can write an Add() method that will be placed into the hwfrmMaint class, and then use Visual FoxPro's object-oriented capabilities to execute this parent method as well as a custom method in the form itself when needed.

Adding a record is pretty easy—you can either do an APPEND and REPLACE or an INSERT. The trick is having the correct default values ready when you perform one of those functions. If you have a single default value, such as today's date for an order date, you can place that expression in the table's Default Value expression in the Table Designer of the data dictionary. However, the other default value, the primary key, might be a bit more difficult. After all, the name of the key will differ according to what table you're in. How do you make it generic?

The answer is that you don't have to. When you add your tables to a VFP database, you can specify default values for each field when a new record is added. All you need is a routine that will generate a unique primary key. Place a call to this routine in your primary key field's Default Value as shown in **Figure 14.16**, and the routine will automatically generate a new key each time a record is added.



**Figure 14.16.** Creating a default value for a primary key.

Then it's just a simple matter of adding a new record with the APPEND command. When this happens, the new record is populated with the default values.

***The Add() method***

The Add() method is somewhat trivial, actually:

```
append blank
thisform.refresh()
```

If you don't include the Refresh() method, the record pointer will be positioned on the new record, but the form will still display the data from the previous record that had been populating the form.

I'll make it lots bigger—three or four times as big!—in the next section.

***INCRID()—a primary key incrementing method***

The basic idea behind this method is to keep a table (I call mine "ITKEY") that holds the last primary key used for every table in the database. When a new primary key value is needed, the routine goes to this table, calculates the next value and stuffs it into the table, and returns that value as the new primary key. The only time this process doesn't work well is if you have a very large number (hundreds per second) of records being added to the same table, because the routine locks the record in the table to ensure that the new primary key is indeed unique.

First things first. You'll need a table with the following structure:

Field Name	Type	Width	Dec	Index	Collate Nulls
CNAKEY	Character	10			No
ILASTKEY	Character	5			No
CADDED	Character	10			No
TADDED	DateTime	8			No
CCHANGED	Character	10			No
TCHANGED	DateTime	8			No

This table stores the last key value used for every key field—primary or candidate—in the system. Each key has its own record, and the record consists of two important fields: the name of the key (cNaKey) and the value of the last key used (iLastKey). The other four fields are just standard audit fields that record the name and date-timestamp of the user who added or last used the field.

When you add a record to a table that requires a primary key, simply call a method named INCRID() that will look in this table for a record that contains the name of the key. (In Figure 14.16, you can see that this method is in a class library—I'll address that later in this chapter when I get rid of MYPROC.)

The call looks like this:

```
0lib.incrid("iidcust")
```

where "iidcust" is the name of the primary key in the table.

If the record is found, calculate the new key, update the record with this value, and then return the new key value.

If you don't find the record, add a new record to the table on the fly, and populate that new record with an initial key value of 1.

As time has passed, I've found the occasion to stuff a value into the primary key table instead of retrieving one. For this reason, I added some extra code that allows the user to pass a second value to the method; this value will be stuffed into the table as the last value used:

```
* incrId()
* if the user passes a value in the second parm,
* we want to stuff that value into ITKEY instead of
* incrementing ITKEY
lpara m.lcNaKey, m.liValueToStuffIntoITKEY
local m.lcNaUser, lcOldWA, llITKEYWasUsed, laBiggestKey, liStartingKey
local llTableWasUsed

if pcount() < 2
  * for backwards compatibility, we assume
  * that the user might not have passed a second parm
  * and if so, we set a flag indicating that
  * we can assume we're just plain old incrementing
  m.liValueToStuffIntoITKEY = -1
else
  * they passed two parms
endif
m.lcOldWA = select()
m.llITKEYWasUsed = used("ITKEY")
if m.llITKEYWasUsed
  sele ITKEY
else
  sele 0
  * note that ITKEY is opened in the default data session!
  Use ITKEY
endif

* check to see if oApp exists
if type("oapp.cnauser") = "U"
  m.lcNaUser = "Interactiv"
else
  m.lcNaUser = oApp.cNaUser
endif

locate for alltrim(upper(cNaKey)) == alltrim(upper( m.lcNaKey ))

if !found()
*
* couldn't find it - create a new record in ITKEY for this key
*
* first, let's see if we can find a table for which this key
* is the PK, and then determine what the biggest value in
* that table is, so that we don't try to put a tiny value into
* ITKEY when the table already has a larger value
m.lcNTable = subs(allt(m.lcNaKey),4)
m.lcNETable = m.lcNTable + ".DBF"
if file(m.lcNETable) or (not empty(dbc()) and indbc(m.lcNaTable, "Table"))
  if used(m.lcNTable)
    m.llTableWasUsed = .t.
    select (lcNTable)
  else
    m.llTableWasUsed = .f.
    select 0
    use (lcNTable)
```

```
endif

* we have the table open
* see if the field exists before we try to do
* a SELECT out of it
* IMPORTANT NOTE!
* if you're selecting a numeric from a table, it
* might be > 10 chars (and an integer, which is what
* we stuff back into ITKEY will only be 10 max -
* so if the numeric value from the table is > 10
* this function will barf
if oLib.FieldExists(m.lcNaKey, m.lcNETable)
    declare laBiggestKey[1]
    laBiggestKey[1] = 1
    select max(&lcNaKey) from (lcNTable) ;
        into array laBiggestKey
    if _tally = 0
        m.liStartingKey = 1
    else
        m.liStartingKey = laBiggestKey[1] + 1
    endif
else
    * field doesn't exist, so init the key
    * why would this happen? Well, the programmer
    * could have passed an incorrectly spelled field name
    m.liStartingKey = 1
endif

* shut the table down if necessary
if m.llTableWasUsed
    * table was already open
else
    use in (lcNTable)
endif
else
    * table doesn't exist, go with a value of 1
    m.liStartingKey = 1
endif
*
messagebox("Creating Key field for " + lcNaKey)
insert into ITKEY (cNaKey, iLastKey, cAdded, tAdded) ;
    values (upper(m.lcNaKey), m.liStartingKey, m.lcNaUser, datetime())
m.iRetVal = m.liStartingKey
else

m.nOldRepr = set("REPROCESS")
set reprocess to 5 seconds
if rlock()
    m.iX = ITKEY.iLastKey
    * check to see that the current value in cLastKey isn't garbage
    if type("m.iX") = "N"
        * hunky dory
    else
        messagebox("The last Key Value is not valid (" + m.iX + ") " ;
        + " Please call your developer with this message")
        return 0
    endif

if m.tiValueToStuffIntoITKEY = -1
```

```
* the user didn't pass a second parm, so we're incrementing
m.iX = m.iX + 1
else
  * the user passed a second parm, so let's use it!
  m.iX = m.tiValueToStuffIntoITKEY
endif

m.iRetVal = m.iX
replace ITKEY.iLastKey with m.iRetVal
replace ITKEY.tChanged with m.lcNaUser
replace ITKEY.tChanged with datetime()

else
  messagebox("Someone else has the Key table locked - try again later")
endif
unlock in ITKEY
set reprocess to (m.nOldRepr)
endif

if m.lITKEYWasUsed
  * it was already open, so leave it open
else
  use in ITKEY
endif

return m.iRetVal
```

Binding a call to this routine to the table's primary key default value has an extra benefit. No matter how you open the table, since it's contained in a database, the Default Value fires when a new record is added with the APPEND command. (You can bypass the Default Value with the SQL INSERT command, but only if you provide a value for any field that has a Default Value rule.)

So what does this mean? It means that the LIB object needs to be instantiated when you append records to the table. Your program would do this automatically, of course, but what about manually? You'd have to do this:

```
set classlib to HWLIB62
oLib = createobject("lib")
use CUST
append blank
```

This can be quite a pain each time you want to add records manually, but it's the price you pay for keeping your primary keys secure.

Let me mention a couple of other subtle points. First, you probably noted the use of "lparameter" and "local," didn't you? Safe programming practices! I should go back and rename the local variables so they conform to the naming convention of using a leading "L," but you know the story: so many variables, so little time. I've still protected myself by declaring the variables as local.

The next major step is to look for the name in ITKEY. Notice that I used LOCATE instead of SEEK. Because there are only a few records in ITKEY, LOCATE is not noticeably slower than SEEK, and I avoid having to maintain an index tag on ITKEY.

The next step is to either create a new record or increment the value of the key found. It is important to note that this new value gets stored in ITKEY and is not rolled back if the user

---

cancels the Add process. In other words, a table might have holes in the primary key field. I make my primary key fields as type Integer, and that provides room for well over a billion values.

I don't roll back unused primary keys because I want to lock the record in ITKEY for as little time as possible. If you wanted to make sure that you never skipped a key, you could lock the ITKEY record until the user saved, which would essentially create a pessimistic locking situation for every table, and that would likely be unacceptable. (You can't decrement the value if the user later decides to cancel, because someone else could have grabbed another key value in the interim.) However, there is one caveat to this technique. While this example demonstrates the use of primary keys, you can easily use INCRID() to return the keys for other fields that also need to be unique and stored in a separate table, such as a customer number. These other types of keys, however, might have to be sequential, such as check numbers. You'll need to do some special coding to handle this type of situation.

You might be wondering why this method isn't in frmMaint instead of in another class? Well, that would mean that you'd only be allowed to get a new key in a form that was subclassed from the hwfrmMaint base class, right? This might not be an accurate assumption, and so—thinking ahead to the time when you might have a different type of form that either adds records or, at least, needs to get new key values—I made a method of another base class.

You could also put the increment ID code into a stored procedure of the database, but you'll have to duplicate that stored procedure in every database you use.

You'll also notice that a property of the application object, oApp.cNaUser, is used to update the ITKEY table with the identity of the last user. The problem with this technique is that oApp is never around when you need it. In other words, what if you want to open the table manually and plunk in a few records? It starts to be quite a pain—now your manual process looks like this:

```
set classlib to HWLIB62, HWAPP62
oApp = createobject("app")
oLib = createobject("lib")
use CUST
append blank
```

And that assumes there aren't any more dependencies involved! Instead, I added code that reduced the cohesion between the two classes—in other words, so that you could use one without having to have the other open as well. I checked to see if oApp.cNaUser was available, and if not, created a dummy value to use. I'll explore this idea further when I discuss components and multi-tier applications.

### A generic Delete() method

If you're going to let the user add records, you're probably going to want to let them delete records, and, as usual, you're not going to want to write that code more than once. The only tough part about deleting is deciding what to do once the user has deleted the record, because you clearly don't want to keep the record pointer on the deleted record.

You have five choices: move the record to the next or previous record, to the top or bottom of the file, or just place it somewhere in the table at random. I've found that the first two choices are the most popular, the next two are somewhat inconvenient, and the fifth choice is popular only with IS departments who hate their users. Personally, I think it makes the most

sense to move to the next record, because that most closely matches the paradigm of a file cabinet: If you get rid of a folder, you see the next one in line. Here's the code:

```
delete
* remember, the table is buffered!
tableupdate()
skip
if eof()
  skip -1
endif
thisform.refresh()
```

Again, like your other generic navigation and maintenance methods, this is found in the Delete() method of your base class hwfrmMaint.

## **Adding hooks to generic methods**

I'm sure you've already realized that these simple methods—Add(), Delete(), Save(), and so on, aren't going to be sufficient for your needs. It seems that you always have to do some special processing some place or another. For example, you might want to fire off a second routine after a record was added, and yet a third routine if the user decided to cancel the Add() in the middle of it. How would you go about this?

You could simply override the Add() method, and write your own custom code. And with a simplistic Add() method like I demonstrated earlier, that wouldn't be too much trouble. But what about the lengthy Save() method? It would be awful, for any number of reasons, to have to copy that method into a subclass merely because you wanted to run another method after the Save routine was finished.

Here's a better way to incorporate flexibility to satisfy these requirements. You'll add several more methods to your maintenance base class and call those methods from your generic code in that same class. However, there won't be anything in those methods in your base class, so the generic method will run just as it always did. The trick is that you can then put code in those methods in your actual forms, and that code will be executed along with your generic code. Here's an example:

1. Add the three following methods to your hwfrmMaint class:

- AddBefore
- AddAfterBeforeSuccess
- AddAfterBeforeFailed

2. Modify your Add() method code like so:

```
m.lAddBeforeWasGood = thisform.AddBefore()  
if m.lAddBeforeWasGood  
    append blank  
    thisform.refresh()  
    m.lAddAfterWasGood = thisform.AddAfterBeforeSuccess()  
else  
    m.lAddAfterFailed = thisform.AddAfterBeforeFailed()  
endif
```

You'll notice that the guts of the Add() method have stayed the same.

3. Next, open the ORD form itself, and put the following message boxes in the methods listed:

```
ORD.AddBefore()  
messagebox("Add Before ")  
  
ORD.AddAfterBeforeSuccess()  
messagebox("AddAfter - Before Success")  
  
ORD.AddAfterBeforeFailed()  
messagebox("AddAfter - Before Failed")
```

4. When you run the form, you'll see that the AddBefore message box displays, then the Append Blank and Refresh commands are executed, and then the AddAfter – Before Success message box displays. You were able to add form-specific code to the Add() method without having to override or duplicate the functionality built into your generic Add() method. Note that I've even provided for the ability to conditionally run the essence of the Add() method, based on whether or not the AddBefore() method returned a true value.



# Chapter 15

## Your First LAN Application: Odds and Ends

As with many types of construction, you can break a job down into two parts. The first part takes 90% of your time, and the last 10% of the job takes another 90% of your time. In this chapter, I'll address a number of "fit and finish" details that make the difference between an app that is thrown together and one with the details taken care of.

The trouble with most books that cover a complex product is that they can only really brush the surface of how to perform certain tasks. Once you start developing, you'll run into a thousand questions regarding the tips and tricks to add polish to your application. Although I don't have a thousand pages to cover each of those questions, I will cover a number of common odds and ends that come up over and over.

### **Adding more functionality to your customer form**

Now that I've covered the basics of a typical data-entry form, it's time to address details that will make this user interface actually useful.

#### **The Sort Order combo box**

As you're navigating through the records in a form, you'll see that the records are displayed in the order that they've been added to the table. Many times, this isn't the order in which you'll want to see them. I've found it handy to provide a combo box on the form (or the toolbar) that lets the user determine in what order the records will be displayed.

There are two pieces to this function: the first is populating the combo box to begin with, because different forms are undoubtedly going to have different sort-order requirements. The second, then, is physically changing the order of the records once the user has made a choice in the combo box.

This is where you're going to see how multiple objects can work together and pass information back and forth; it's quite a change from the procedural code you're probably used to. There are two objects here—the toolbar and the form—and each has its own responsibilities and knowledge. The toolbar, for example, will be responsible for displaying the available sort orders and responding to a user's change in the sort order to display.

However, because the toolbar is a generic object, it won't know which sort orders are available for a specific form—indeed, not even the form's base class will know that. After all, that information is specific to an actual form. The form should be responsible for actually changing the sort order as well, because the table is bound to the form, not to the form class or the toolbar.

As you work through this, keep in mind that this is still an example. In a real application, you'd most likely keep the available sort orders for a table in a data dictionary, not hard-coded in the form.

### Populating the combo box in the toolbar

The basic idea behind populating the toolbar combo box is to store the available sort orders (and their tags) in an array property of the form, and then upon activation of the form, stuff that information into the aItems property of the combo box. Why upon activation? If you've got several forms open, you'll want the toolbar to reflect the sort orders for the active form—not, for example, just the last one that was opened. Thus, as each open form is brought forward (activated), the combo box will be repopulated with new sort orders.

The Activate event of the form class can then handle stuffing sort orders into the toolbar combo box's aItems property. The following code does so by simply iterating through the aSortOrders property of the active form:

```
hwfrmMaint.activate()
m.lnRows = alen(_screen.activeform.aSortOrders,1)
dime oApp.oToolBar.hwcboSort.aItems[m.lnRows,2]
for m.li = 1 to m.lnRows
  oApp.oToolBar.hwcboSort.aItems[m.li,1] = _screen.activeform.aSortOrders[m.li,1]
  oApp.oToolBar.hwcboSort.aItems[m.li,2] = _screen.activeform.aSortOrders[m.li,2]
next
oApp.oToolBar.hwcboSort.requery()
oApp.oToolBar.hwcboSort.DisplayValue = oApp.oToolBar.hwcboSort.aItems[1,1]
```

Note that if you use more than one form with the toolbar, the last line will change the order of the table when you switch to another form and then come back to this one. You may want to add extra code to handle this situation if it's important to you.

However, the initialization of which sort orders are actually available still has to be done in each individual maintenance form. Because it has to be done manually, this means it could be forgotten, which is a possibility that has to be trapped. Ideally, the user should never see any evidence of this happening. To make this transparent, I've created a method in the hwfrmMaint class, InitSortOrders(), that should only be used by the developer to populate the form's sort order array, aSortOrders. If they do their job right, the code in InitSortOrders() would look like this:

```
ORD.InitSortOrders()
dime thisform.aSortOrders[4,2]
thisform.aSortOrders[1,1] = "Last Name - First Name"
thisform.aSortOrders[1,2] = "ucnalf"
thisform.aSortOrders[2,1] = "Company Name"
thisform.aSortOrders[2,2] = "ucnao"
thisform.aSortOrders[3,1] = "Country"
thisform.aSortOrders[3,2] = "uccountry"
thisform.aSortOrders[4,1] = "Physical Order"
thisform.aSortOrders[4,2] = ""
```

Then, in the Init() of the form, this method is called. If the programmer did their job, this method will return True, and the Init() will be done. What if the programmer forgot to populate ORD.InitSortOrders()? I added one line of code to the class definition's InitSortOrders():

```
HwfrmMaint.InitSortOrders()
return .f.
```

---

Thus, the Init() method populates the array either with a dummy “Physical Order” value or with real values:

```
HwfrmMaint.Init()
if !empty(thisform.cToolBar)
oApp.ShowAppToolBar( thisform.cToolBar )
if thisform.InitSortOrders()
else
  dime thisform.aSortOrders[1,2]
  thisform.aSortOrders[1,1] = "Physical Order"
  thisform.aSortOrders[1,2] = ""
endif
endif
```

The Init() fires only once, while the Activate() fires every time the form is brought to the forefront of the application. Thus, the form’s array is populated only once, while the toolbar’s array gets populated whenever it needs to be.

### Changing the order of the records

So far, so good. The form and the toolbar both have the data they need. But users don’t care about any of this—they just want to select a sort order in the toolbar’s combo box and see the order change in the form. The Anychange() method of the combo box (this is a custom method that is called from both the native Interactive Change and Programmatic Change events, and was described in Chapter 10), just calls a method of the active form:

```
hwtrnav.hwcboSort.Anychange()
_screen.activeform.ChangeSortOrder()
```

The active form’s ChangeSortOrder() method, then, looks at the DisplayValue in the combo box, gets the associated tag, and changes the order of the table accordingly:

```
hwfrmMaint.ChangeSortOrder()
* find out what they selected
* get the actual tag (second column)
* set the order to the actual tag
m.liElementNumber = ascan(oApp.oToolbar.hwcboSort.aItems,
oApp.oToolbar.hwcboSort.DisplayValue)
m.lcTag = oApp.oToolbar.hwcboSort.aItems[m.liElementNumber + 1]
set order to (lcTag)
```

An alternative method would be to pass This.Value to the ChangeSortOrder() method. The BoundColumn of the combo box would have to be set to 2, but then the all of the code in the ChangeSortOrder() is unnecessary, except for the last line that uses that parameter.

You might again be wondering, “Why not just stuff the ChangeSortOrder() code into the toolbar itself?” Just as the other buttons in the toolbar call methods in the form class, so should this one. After all, you might need to override (or create generic hooks into, as described in the previous section in this chapter) a particular form’s ChangeSortOrder() code.

Obviously, there’s no error trapping here; you might want to check for the existence of the tag before setting order to it willy-nilly. But the purpose here was to show you how to put code

in the proper place when you have multiple objects working together. What seems to be a fairly customizable routine has now been made generic; only the data supplying the routine is distinct.

## **Concepts for child forms**

The next task to cover is that of using child forms to add and edit child data and to provide a query interface. Before you start building those data-entry and query forms, you'll need to know some background.

### **Passing parameters to a form**

The first thing you'll want to know is how to pass a parameter to a form. There are actually two pieces: the syntax used when calling the second form and the mechanism inside the second form that accepts the parameter. As you know, the syntax to call form X is:

```
do form X
```

If you want to pass a parameter to form X, simply include a “with” clause, like so:

```
do form X with m.cMyParm
```

If you want to pass more than one parameter, separate the parameters with commas, like so:

```
do form X with m.cMyParm1, m.cMyParm2, m.cMyParm3
```

Then, in form X, you'll need a parameters statement in the Init() method, like so:

```
* myform.init()  
lparameter m.tcMyParm1, m.tcMyParm2
```

From here, you can manipulate the parameters much like you would in a procedure or function. Many people make the mistake of trying to accept the parameters in the Load() method, because Load() fires before Init(), but the form doesn't actually exist in memory at the beginning of the Load() event, so there's nothing ready to accept. By the time the form's Init() event fires, the form and its controls have been created and are ready for business.

One thing to keep in mind is that the parameters are scoped local to the Init() method—they're not available to the entire form unless you do something special. Typically, that “something special” would be to assign the parameter values to form-level properties, like so:

```
* myform.init()  
lparameter m.tcMyParm1, m.tcMyParm2  
thisform.cMyProperty1 = m.tcMyParm1  
thisform.cMyProperty2 = m.tcMyParm2
```

## Returning values from a form

The second thing you'll want to know is how to return a value from a child form to the calling form. This, too, involves two pieces and again is not as straightforward as you'd like. First, structure the call to the form to accept a return value as well, like so:

```
do form X to m.cMyReturnValue
```

If m.cMyReturnValue doesn't already exist, it will be created for you. Next, add code to the Unload() event of the child form to return a value, like so:

```
* childform.unload()  
return m.cValueToReturn
```

So far, so good. However, you'll spend an hour or two tearing your hair out if you try the following code in your Unload() event:

```
* childform.unload()  
return thisform.txtName.value
```

You can't reference a value of a member of the form, such as the value property of a control, in the Unload()—by the time the return command fires, the form has already begun self-destructing and the controls don't exist anymore. Thus, you'll have to assign the property to a local variable and send that variable back, like so:

```
* childform.unload()  
m.lcTempVariable = thisform.txtName.value  
return m.lcTempVariable
```

Finally, note that a form must be modal to return values in the Unload() event.

## Passing object references

Now that you've got that down, you're probably wondering how to return more than one value to the calling form. Since you could simply pass a string of parameters to a child form, you might have already tried to return a series of values, like so:

```
* childform.unload()  
m.lcTempVariable1 = thisform.cPropertyValueToReturn  
m.lcTempVariable2 = thisform.textbox1.value  
return m.lcTempVariable1, m.lcTempVariable2
```

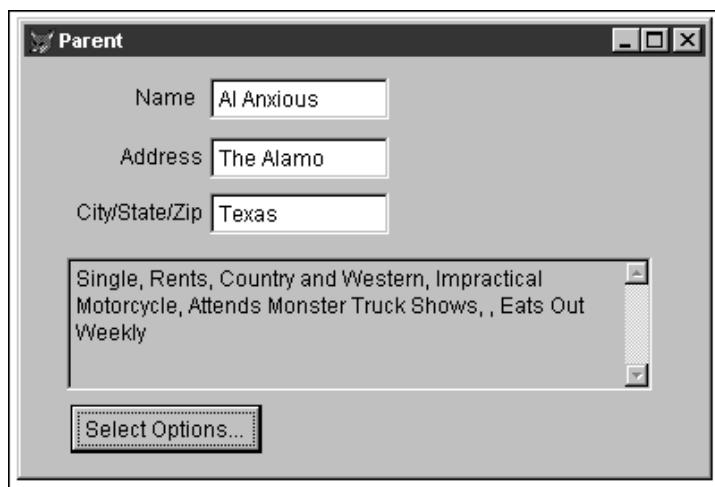
Unfortunately, this technique doesn't work, so it's back to the drawing board.

There are several "workarounds" that programmers use to perform this function; I think the cleanest and "most proper" is to simply pass an object reference of the calling form to the child form, and then have the child form actively populate properties as needed. In other words, instead of passing a huge chunk of data back to the calling form, simply have the child manipulate the parent directly.

For example, suppose that you have a child form that allows the user to make a number of choices, such as setting attributes in check boxes and option buttons, as shown in **Figure 15.1**. (The Select Options button in **Figure 15.2** opens this form.) The calling form simply needs to display the result of those choices in a concatenated text string, as shown in Figure 15.2.



**Figure 15.1.** This child form allows the user to select a number of attributes that need to be sent back to the calling (parent) form.



**Figure 15.2.** The string in the edit box is concatenated from option-button and check-box selections made in the child form.

 Obviously, the child form can't pass back the selected values of each control to the parent. And concatenating the string in the child, while possibly workable for this specific example, isn't necessarily a valid technique in general. Here's how this is done. (The source code for these two forms is included with the rest of the CH14D files from the last chapter.)

First, the parent form, CALLER.SCX, contains the following properties:

- 
- cMarriedSingle
  - cOwnsrents
  - cRockCountry
  - cSensibleImpractical
  - lAttends
  - lOwns
  - lEats

The parent form also contains the following call to the child form, CALLERCHILD.SCX, in the Click() event of the Select Options command button:

```
* SelectOptions.Click()
do form callerchild with thisform
```

This command passes an object reference—a reference to the parent form itself—as a parameter to the child form. Then, the Init() of the child form contains the following code, in order to accept the parameter being passed:

```
* CallerChild.Init()
lpara toCaller
thisform.oCallingForm = toCaller
```

The child form obviously has a property, oCallingForm, that contains the object reference—it's just the name of a variable—and the parameter is stored to this property. Now the child form can reference objects (properties, and even methods!) of the calling form, with syntax like this:

```
thisform.oCallingForm.cMarriedSingle = "Some Value"
```

The entire Click() method of the Done button in the child form assigns the values of the option buttons and the check boxes on the form to their respective properties in the parent form (note that I took the easy way out and didn't name every option group or check box for this example).

```
* Done.Click()
thisform.oCallingForm.cMarriedSingle = thisform.optiongroup1.value
thisform.oCallingForm.cOwnsRents = thisform.optiongroup2.value
thisform.oCallingForm.cRockCountry = thisform.optiongroup3.value
thisform.oCallingForm.cSensibleImpractical = thisform.optiongroup4.value
thisform.oCallingForm.lAttends = thisform.check1.value
thisform.oCallingForm.lOwns = thisform.check2.value
thisform.oCallingForm.lEats = thisform.check3.value
thisform.release
```

Once you get the hang of this technique, you can go a step further by creating a separate object—a parameter object—onto which you can hang your parameters. Then you can pass an object reference of this parameter object from one form to another.

With these tricks in your back pocket, you’re now ready to handle a variety of child forms for the Order Maintenance form.

## Adding a list box to display child records

Among the many religious wars that programmers fight (Windows vs. Linux? Mac vs. PC? Pessimistic vs. Optimistic locking? Coke vs. Pepsi?) lies the question of whether to use grids for displaying and editing data in child tables.

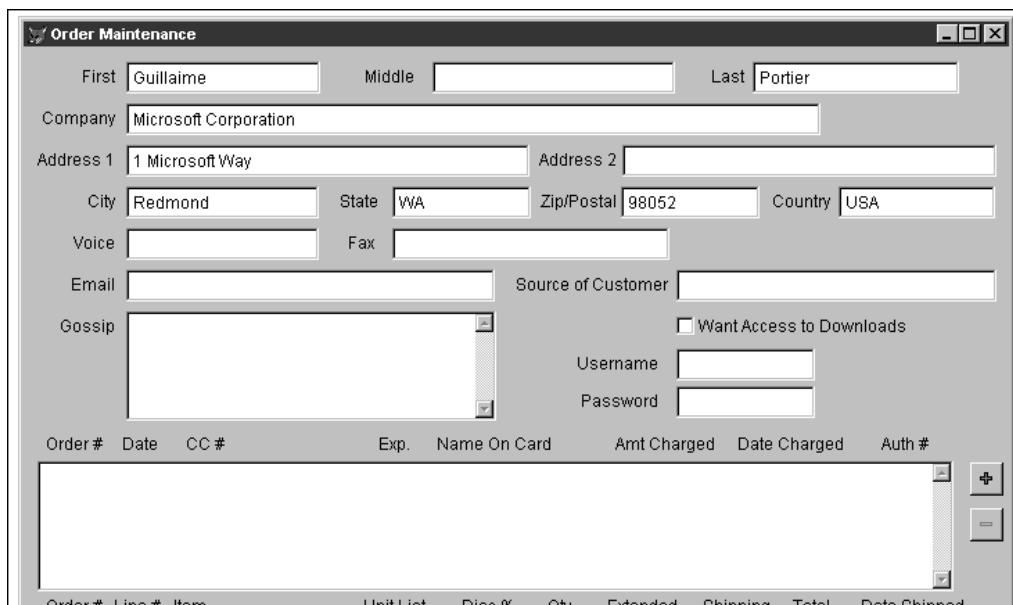
Jim Booth and Steve Sawyer have dedicated a fair amount of space to this argument in their book, *Effective Techniques for Application Development*, and I won’t repeat them. Instead, I’ll ignore the question altogether, and get you prepared for the fight by showing you how to display child records in a list box. Once you’re comfortable with this idea, you’ll see some shortcomings and you can then decide if you want to graduate to grids.

There are four parts to using a list box for child records. The first is simply the population of the list box: grabbing and formatting all the child records for a particular parent. This has to be done not only during navigation, but also during the addition of new parent records (where there likely won’t be any children).

The other three parts have to do with adding more children, deleting children, and editing existing children. These three, obviously, are more easily separated from the rest of the form, because they can be formatted in discrete methods that are called via command buttons or hotkeys. I prefer to use a pair of command buttons for adding and deleting, as shown in **Figure 15.3**, and call the Edit method by double-clicking on the row to edit. Notice that the image for the Delete button is dimmed—this feature is known as a “visual clue” and indicates that the button is not currently available, because there are no child records to delete.

### Populating a childbearing list box

It should probably be obvious to you by now that a separate method is needed to fill the list box with the data from the child records. However, you still might not be sure whether to put this method in the form or in the form class. In this case, the answer is the form itself, because not every maintenance form will have a child list box, and other maintenance forms might have several child list boxes—or even (as this one eventually will) multiple list boxes representing children and grandchildren.



**Figure 15.3.** Two command buttons to the right of the list box can be used for adding and deleting rows from a child list box.

This method will be called each time the form is refreshed: when you move back and forth through the records, when you add a new record, or when you delete a record (and thus have to move to a different parent). It will also be called each time you add, edit, or delete a child record, because the data in the list box again might have changed.

To get started before you start slinging code, add the two command buttons on the side of the list box. I used the hwtbrButton class because the buttons are the right size, and, conceptually, you could see how these two buttons could migrate to a toolbar at some point.

1. Add two command buttons using the hwtbrButton class.
2. Change the names of the buttons to hwtbrButtonOrderAdd and hwtbrButtonDelete.
3. Change the picture properties like so:

```
hwtbrButtonOrderAdd.Picture: ="add_gr.bmp"
hwtbrButtonOrderAdd.DisabledPicture: ="add_grdis.bmp"
hwtbrButtonOrderDelete.Picture: ="del_gr.bmp"
hwtbrButtonOrderDelete.DisabledPicture: ="del_grdis.bmp"
```

A subtle point to remember here is that if you select the bitmap by using the ellipsis button in the Properties window, you'll see the full path displayed in the item's entry. However, the .SCX or .VCX only stores a relative path.

You'll also notice that each button has two pictures associated with it. One will be displayed if the button is enabled (because, in the case of Add, you are permitted to

add a record, or, in the case of Delete, you can delete certain records in the list box) and the other if the button is disabled. The code I'll show you in a minute will enable and disable the buttons; Visual FoxPro will take care of displaying the proper bitmap according to the Enabled status of the button.

4. Add the table that contains the child data to this form's Data Environment. In this case, it's the ORDERH (Order Header) table.
5. Add a method that will be called to fill the list box, such as FillOrderListbox(). Don't use a generic method name (like "FillList") because sure as shooting, you're going to end up with a second list box on the form if you do.

Here's the code for the FillOrderListbox() method:

```
* FillOrderListBox()
select "", iNoOrd, dOrd, cNoCC, cExp, cNaCC, nAmtCharge, dCharged,
cNoAuth ;
  from ORDERH ;
  where ORDERH.iidCust = thisform.iid ;
  into array laOrderH
if _tally = 0
  dime thisform.hwlstOrders.aItems[1,9]
  thisform.hwlstOrders.aItems[1,1] = "<No Orders>"
  thisform.hwlstOrders.aItems[1,2] = 0
  thisform.hwlstOrders.aItems[1,3] = {}
  thisform.hwlstOrders.aItems[1,4] = ""
  thisform.hwlstOrders.aItems[1,5] = ""
  thisform.hwlstOrders.aItems[1,6] = ""
  thisform.hwlstOrders.aItems[1,7] = 0.00
  thisform.hwlstOrders.aItems[1,8] = {}
  thisform.hwlstOrders.aItems[1,9] = ""
else
  * we've got an array
  dime thisform.hwlstOrders.aItems[alen(laOrderH),1,9]
  for m.li = 1 to alen(laOrderH,1)
    thisform.hwlstOrders.aItems[m.li,2] = laOrderH[m.li,2]
    thisform.hwlstOrders.aItems[m.li,3] = laOrderH[m.li,3]
    thisform.hwlstOrders.aItems[m.li,4] = laOrderH[m.li,4]
    thisform.hwlstOrders.aItems[m.li,5] = laOrderH[m.li,5]
    thisform.hwlstOrders.aItems[m.li,6] = laOrderH[m.li,6]
    thisform.hwlstOrders.aItems[m.li,7] = laOrderH[m.li,7]
    thisform.hwlstOrders.aItems[m.li,8] = laOrderH[m.li,8]
    thisform.hwlstOrders.aItems[m.li,9] = laOrderH[m.li,9]
    thisform.hwlstOrders.aItems[m.li,1] =
      = right(str(laOrderH[m.li,2]),4) ;
      + " " + dtoc(laOrderH[m.li,3]) ;
      + " " + laOrderH[m.li,4] ;
      + laOrderH[m.li,5] ;
      + laOrderH[m.li,6] ;
      + str(laOrderH[m.li,7],10,2) ;
      + dtoc(laOrderH[m.li,8]) ;
      + laOrderH[m.li,9]
  next
endif
thisform.hwlstOrders.requery()
thisform.hwlstOrders.DisplayValue = thisform.hwlstOrders.aItems[1,1]
```

---

So far, so good. But as of now, this is just a dumb method—it doesn’t get called from anywhere. Time to fix that next. This could be done manually from each place that the form changes data, such as the Add() method, the Next() method, and so on. But what’s common to each of these? They’re all going to require a refresh of the form’s data. So why not just call the method from the form’s Refresh() method?

```
* ORD.refresh()  
thisform.fillOrderListBox()
```

In this particular case, it will work nicely. You want to be careful, however, because Refresh() can be a time-consuming process—you don’t want to call the form’s Refresh() method any old time you want a little bit of the form to be updated.

If you’re following along, and you’ve tried this, you’ll see that there’s one more missing piece—a very important piece. The SELECT command in the FillOrderListBox() method grabs all of the child records according to a “ORDERH.iidCust = thisform.iid” WHERE clause, and I’ve not mentioned anything about the “thisform.iid” property yet.

This is a little secret of mine to make my life easier. In my maintenance form class, hwfrmMaint, I create a property called iid, and use it to hold the primary key of the current record. I find it’s useful in many respects, and this one—where I want to know the key of the current record in order to find children—is a perfect example. To use it, and to keep things as generic as possible, I create a second property in the hwfrmMaint class that holds the name of the primary key field, cNamePK.

Then, in the Init() of the form (not the form class!), I actually assign the name of the Primary Key to this property, like so:

```
* ORD.Init()  
thisform.cNamePK = "iidCust"  
dodefault()
```

Afterwards, of course, I call the parent class method with the DODEFAULT() function. I call DODEFAULT() after I assign the property because I might need the new property value in the parent code. Now, all that’s left is to update the property each time the form is refreshed. I do this in the Refresh() method of the form class. Notice how using the cNamePK property of the form class means I can keep my code as generic as possible.

```
* HwfrmMaint.refresh()  
thisform.iid = eval(thisform.cNamePK)
```

Now that I’ve got code in the form class’s Refresh() method, it means that I have to change the ORD.refresh() method as well:

```
* ORD.refresh()  
dodefault()  
thisform.fillOrderListBox()
```

This time, I call the parent class method first. There could be generic refresh code that I’d want to execute before I call any custom code, like the FillOrderListBox() method.

One final note regarding navigation: The Add and Delete buttons need to be handled properly. If you're on a parent record without children, the Delete button needs to be dimmed, but it needs to be enabled if you move on to a record that has children (which, clearly, could be deleted). The Add button needs to be disabled during the process of adding a new parent.

I created a form-level method called HandleOrderButtons, and called it at the end of the FillOrderListBox() method. Here's the last few lines of the FillOrderListBox() method:

```
endif  
thisform.hwlstOrders.requery()  
thisform.hwlstOrders.DisplayValue = thisform.hwlstOrders.aItems[1,1]  
thisform.HandleOrderButtons()
```

The HandleOrderButtons() method looks like this:

```
* HandleOrderButtons()  
* disable the Add button if there isn't a person or company name  
* (meaning we're probably adding a record)  
if empty(thisform.hwtxtFirst.value) ;  
  and empty(thisform.hwtxtLast.value) ;  
  and empty(thisform.hwtxtCompany.value)  
  thisform.hwtbrButtonOrderAdd.enabled = .f.  
else  
  thisform.hwtbrButtonOrderAdd.enabled = .t.  
endif  
  
* disable the Delete button if there are no children  
if thisform.hwlstOrders.DisplayValue = "<No Orders>"  
  thisform.hwtbrButtonOrderDelete.enabled = .f.  
else  
  thisform.hwtbrButtonOrderDelete.enabled = .t.  
endif
```

Once the record has been saved, of course, the Add Orders button will need to be enabled, and that's taken care of when the Save() method calls the form's Refresh() method.

### **Adding children**

The basic strategy behind adding child records, when using a list box to display the children in a form, is to pop open a form to add the child record, and shut down the form after saving the child. Because this type of process is fairly common, and because it has a lot of standard code associated with it, it makes sense to create a class for the child form.

Before that, however, I've changed the hwfrm base form class to have a Done() method, moved the Done button's Click() code to that Done() method, and then called the Done() method from the Done button's Click() method. It's actually more work right now, but it will become useful shortly.

1. Create a form class, hwfrmChild, based on the hwfrm class.
2. Change the following attributes of the hwfrmChild class:
  - Caption: hwfrmChild
  - Window type: 1-modal

(Remember? A form must be modal to return values in its Unload() event.)

3. Add three properties:

- iid (contains the ID of the current record in the child form)
- iidParent (contains the ID of the child record's parent)
- lDidAdd (contains a flag indicating whether the user added a record or canceled out of the child form)

4. Add the following code:

```
* hwfrmchild.init()
lpara m.tiidParent
thisform.iidParent = m.tiidParent

* Hwfrmchild.Activate()
append blank

* hwfrmchild.Unload()
m.lDidAdd = thisform.lDidAdd
return m.lDidAdd

* hwfrmchild.Save()
=tableupdate()
thisform.lDidAdd = .t.
thisform.release

* hwfrmchild.Done()
thisform.lDidAdd = .f.
dodefault()
```

Some of this already looks familiar—you'll notice how I'm returning the value of the form's lDidAdd property to the parent form via the Unload() event, for example.

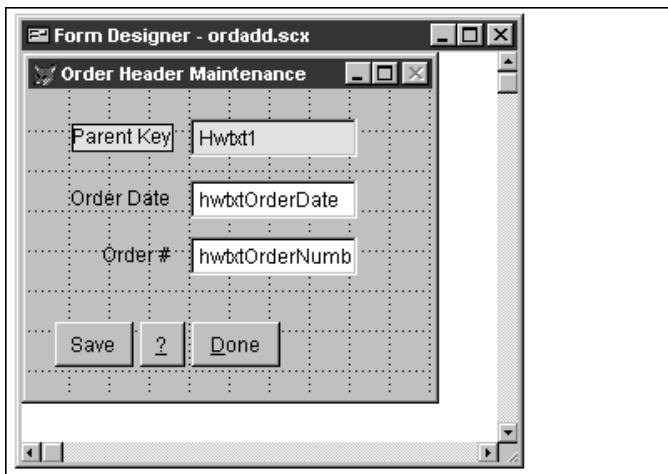
5. Drop a Save button on the form and add the following code to its Click() method:

```
* Save button click:
thisform.save()
```

You'll notice that this is all generic code. It handles a great deal of the processing so you have to write as little custom code as possible in the actual child form. Now it's time to create a form from this class:

1. Create a form named ORDADD.SCX.
2. Change the following attributes:
  - Name: ORDADD
  - Caption: Order Header Maintenance
3. Add the ORDERH table to the data environment.

4. Add a few text boxes and labels to the form, using the hwtxt and hlbl base classes, as shown in **Figure 15.4**. Ignore the fact that the top two controls, the Parent Key label and the hwtxt1 text box, look different in the figure for the time being. Change the text box control names as shown.



**Figure 15.4.** The Order Header Maintenance form in design mode.

Obviously, a real form would have more controls, but these few serve well enough as an example.

5. Map the text box control sources to the appropriate ORDERH fields (dOrd and iNoOrd).
  6. Add the following code:
- ```
* Ordadd.activate()
dodefault()
thisform.hwtxt1.value = thisform.iidParent
```
7. Open the ORDERH table and create a Default Value for the iidorderh primary key field:
 

```
olib.incrid("iidorderh")
```

 And that's it! The rest of the code that handles adding and saving data is in the hwfrmChild class!
  8. Finally, call the child form from the Order Maintenance form's OrderAdd() method and handle the results:

```
* ORD.OrderAdd()
do form ORDADD with thisform.iid to m.l1DidAdd
  sele CUST
  if m.l1DidAdd
    * need to refresh the form's list box
    thisform.FillOrderListBox()
  else
    * user cancelled out of the add
  endif
```

You'll see that the primary key of the current record in the Order Maintenance form (in this case, the ID of the current customer) is passed to the Add Order form to be used as a foreign key in the Order Header record. And the child form returns a logical value depending on whether a record was actually added. This logical flag is then used to determine whether or not the Order Maintenance form's Orders list box needs to be refreshed. This functionality is not absolutely necessary, but it's nice to have and could speed up performance in some cases.

### **Editing an existing child record**

You can edit an existing child record in one of two ways:

- Pass the ID of the child and then select the data from the table.
- Pass all of the values from the array.

In either case, you could return whether the record was edited.

Let me give you another idea. You've already got a maintenance form for adding a record. Why not use the same form for editing as well? What would be required to do so?

First, you'd want to be able to handle both adding and editing, and the form would have slightly different behavior depending on which you're doing, so you'd want to pass a flag to the child form that indicates whether you're adding or editing. The code in the form's Init() would look like this:

```
* HwfrmChild.Init()
lpara m.tiidParent, m.tiidChild, m.lAddNotEdit
thisform.iidParent = m.tiidParent
thisform.iid = m.tiidChild
thisform.lAddNotEdit = m.lAddNotEdit
```

Assuming that you're editing from the table (as opposed to array values), you'll also want to pass the ID of the child record to edit. You're not going to need a child ID if you're adding, so perhaps you could pass a parameter with value zero if you're adding a record. You'll want properties for both the AddNotEdit flag and the child ID value.

Next, inside the form, you'd either populate the form with a blank record (through APPEND BLANK) or with live data (by positioning the record pointer in the table to the record being edited). You could do this in the hwfrmChild's Activate() event, like so:

```
if thisform.lAddNotEdit  
    append blank  
else  
    * code to position record pointer on record to edit  
endif
```

In both cases, the data for the record (either blank values or values to edit) will be displayed in the controls on the child form.

Finally, you'll need to have code in the Order Maintenance form's OrderEdit() method. This will be called from the list box's DblClick() method and would look something like this:

```
m.liElementNumber = ascn(thisform.hwlstOrders.aItems, ;  
    thisform.hwlstOrders.DisplayValue)  
if m.liElementNumber = 0  
    messagebox("Unable to find " ;  
        + chr(13) + thisform.hwlstOrders.DisplayValue ;  
        + chr(13) + " in the list box.")  
else  
    m.liidOrderH = thisform.hwlstOrders.aItems[m.liElementNumber + 1]  
    do form ORDMAINT with thisform.iid, m.liidOrderH, .f.  
endif
```

First, you need to determine which row in the list box is highlighted so that you can tell which row is supposed to be edited. No matter what the occasion, I always test the result of the ASCAN function to make sure a valid element number was returned—if for some reason the Display Value wasn't found in the array (most likely a programming error), it's good to alert the user. Notice that instead of some “Error Occurred” type of message, I took a bit of extra trouble in this message box to explain what happened, and included the specific data as well.

An alternative method to this business of ASCAN is to use the ListIndex property of the list box:

```
m.liElementNumber = thisform.hwlstOrder.ListIndex
```

The code in the ELSE segment then determines the primary key for the record to be edited, and calls a dual-purpose ORDMAINT form with the proper parameters as described earlier.

The source code for this section includes this code but doesn't include the child form, ORDMAINT.SCX, itself.

### **Deleting an existing child record**

Deleting a record is fairly easy, compared to these other processes. A little bit of housekeeping is needed in order to keep things tidy, but that's all. Here's the code in the ORD.OrderDelete() method, which, of course, is called from the Click() method of the Delete Order command button:

```
m.liElementNumber = ascn(thisform.hwlstOrders.aItems,  
    thisform.hwlstOrders.DisplayValue)  
if m.liElementNumber = 0  
    messagebox("Unable to find " ;
```

```
+ chr(13) + thisform.hwlstOrders.DisplayValue ;
+ chr(13) + " in the list box.")
else
m.l1WantToDelete = messagebox("Are you sure you want to delete " ,
+ chr(13) ;
+ chr(13) + thisform.hwlstOrders.DisplayValue ;
+ chr(13) ;
+ chr(13) + "now?", MB_YESNO+MB_ICONQUESTION +MB_DEFBUTTON2 )
if m.l1WantToDelete = IDYES
m.liidOrderH = thisform.hwlstOrders.aItems[m.liElementNumber + 1]
sele ORDERH
delete for ORDERH.iidOrderH = m.liidOrderH
sele CUST
wait wind nowa "Deleted"
else
wait wind nowa "Delete canceled"
endif
endif
thisform.FillOrderListBox()
```

Just as with editing a record, I do a bit of error trapping to make sure that the row is valid. The rest of the code—in the ELSE segment—asks users to confirm that they do, truly, want to delete the row, and then performs the deletion if appropriate. Finally, the method repopulates the list box because the contents have changed.

## Odds and ends

### Checking for `_screen.activeform`

In much of the application, I reference “`_screen.activeform`” when making a call to a method. If you’ve been following along on your machine, you might have noticed the error message “ACTIVEFORM is not an object” if you’ve tried to click on a toolbar or run a menu option when windows aren’t open or when debug windows are active. These windows participate in the event loop, but they aren’t really part of the program. As a result, when the `_screen.activeform.next()` method is executed, Visual FoxPro either fails trying to find “`activeform`” or tries to find the `Next()` method attached to a window that doesn’t have one.

The solution is to provide a protective wrapper for each call to a method that belongs to a form. This wrapper will examine the type of the active form, and if it’s not “O” (it’s not an object), the method won’t be allowed to fire. For example, the `Click()` method of the Next toolbar icon that contains this code:

```
_screen.activeform.next()
```

will be changed to

```
if type("_screen.activeform") = "O"
    _screen.activeform.next()
else
    wait window "You can't use this function unless an application form is active"
endif
```

The Next() method is also called in the Record menu. This will require two changes. First, change the Record menu options to a Procedure result; second, enter the same code as is in the corresponding toolbar icon's Click() method:

```
if type("_screen.activeform") = "O"
  _screen.activeform.next()
else
  wait window "You can't use this function " ;
    + "unless an application form is active"
endif
```

This will have to be done for all of the navigation and maintenance menu options in the system: Next, Back, Add, Delete, Find, List, Save, and Undo.

## **Dimming the Record menu pad**

You'll notice that the Record menu pad—and its menu options—are all available even when there is no active form. This is a problem because you can select one of the menu options and it will bomb, since there is no active form. It is also less than kind to the user, because the Record menu pad should just be completely disabled when it isn't appropriate. It shouldn't disappear, however, because that might be confusing to the user.

This is a problem with a very simple solution. Just check the value of oApp.nFormInstanceCount (this is the same property used when figuring out whether or not to activate the toolbar) in the Record menu pad's SKIP FOR expression:

1. Modify the IT.MNX menu.
2. Select the Record menu pad.
3. Select the Options checkmark command button.
4. Select the Skip For edit box.
5. Enter the expression:  
`oApp.nFormInstanceCount < 1`
6. Close the dialog, save the menu, and rebuild it.

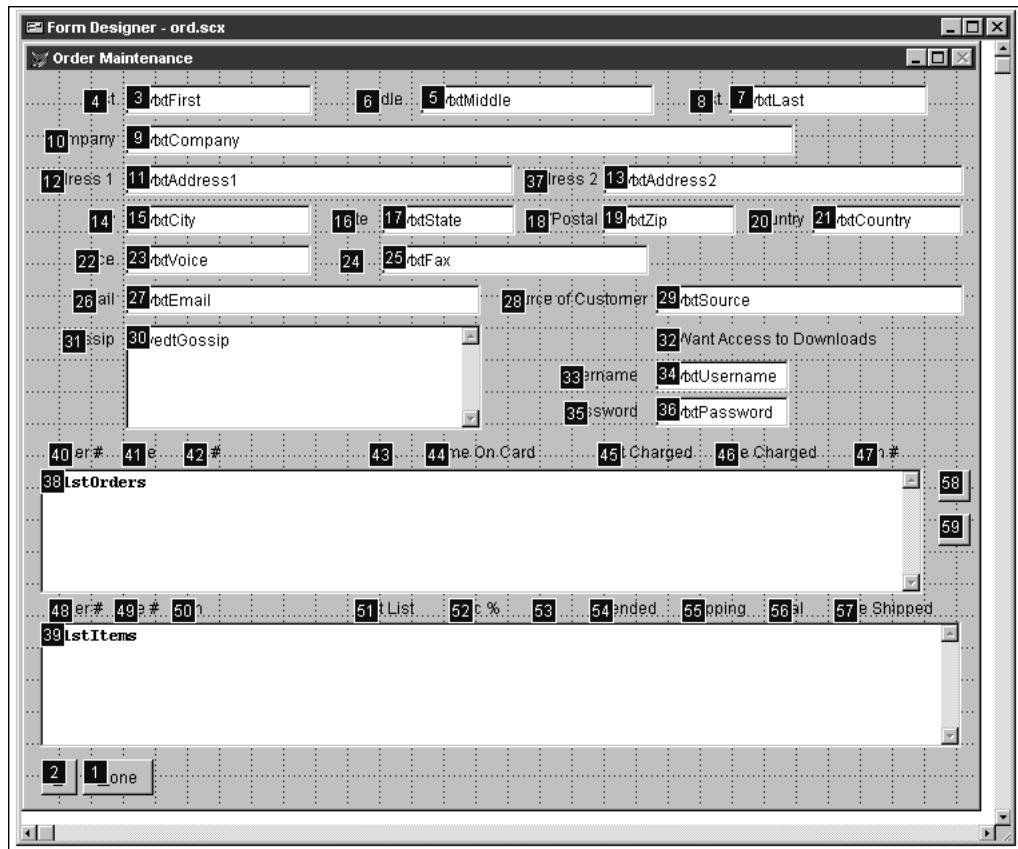
When you run the app, the Record menu will be dimmed until you open a form.

## **Setting tab order on a form**

You probably have noticed that, on a form, the focus moves from one control to another in the order that they were put on the form. This order is called the "tab order." And the tab order is most likely not the order you want. You can change the tab order through two similar mechanisms; which depends on the choice of the Tab Ordering combo box in the Forms tab of the Tools, Options dialog.

If tab ordering is set to Interactive, you can open a form and then select the View, Tab Order menu command. (The Tab Order menu command isn't displayed if you don't have a

form open.) Doing so will display numbered boxes over the upper left corner of each control on the form, as shown in **Figure 15.5**.

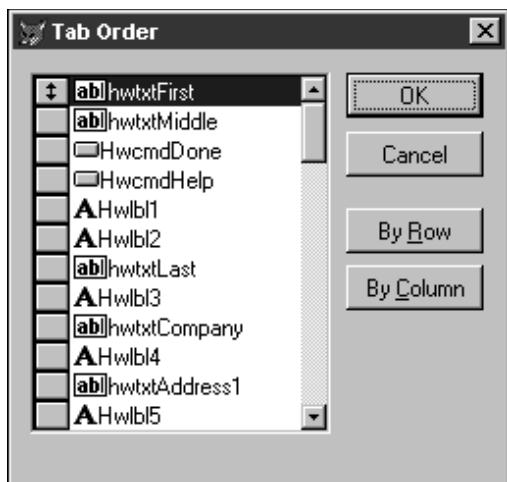


**Figure 15.5.** The numbered boxes show the order in which focus will move on the form.

To change the tab order, just hold down the Ctrl key and click the controls in the order in which you'd like focus to move. You can, of course, ignore controls that don't ever get focus, such as labels.

If tab ordering in Tools, Options is set to By List, selecting the View, Tab Order menu command opens the Tab Order dialog as shown in **Figure 15.6**.

Unless you've got a small number of controls, this mechanism is awkward to use because it holds only a dozen controls and the dialog can't be resized. I prefer to use the Interactive ordering to set the tab order, and then use the Tab Order list to make minute adjustments, such as swapping the order of two controls.



**Figure 15.6.** The Tab Order dialog allows you to use a list box with mover bars to adjust the tab order of controls on a form.

## Creating OLIB (ridding MYPROC.APP)

 It feels sort of old-fashioned to have a procedure file hanging around in a Visual FoxPro application, and, indeed, we don't really need to have one anymore. You can move the functions in a proc file into methods of a class library. Not only does their functionality remain available to the application, but doing so provides additional flexibility in the event that a function needs to be subclassed in the future. (The source code for this section is in the CH15 directory.)

I created a class called LIB and stored it in a class library called HWLIB62 in the last section. There's only one method in it right now—the INCRID() routine that is used to generate primary key ID values. But it's little work to move all of the other functions in MYPROC to the LIB class.

First (assuming you've already got the LIB class in HWLIB62), create methods for each of the functions in MYPROC, and move the code into the library, as shown in **Figure 15.7**.

Next, you'll want to make sure that the oLib object actually exists:

```
set classlib to HWAPP62.VCX, HWCTRL62.VCX, HWLIB62.VCX
oEnv = createobject("ENV")
oApp = createobject("APP")
oLib = createobject("LIB")
```

And, of course, you can get rid of the call to MYPROC in IT.PRG as well. Finally, you'll need to replace calls to the functions in MYPROC with calls to methods in the LIB class. Thus, a call like so:

```
m.nAmtRAM = GetRAM()
```

would be replaced by:

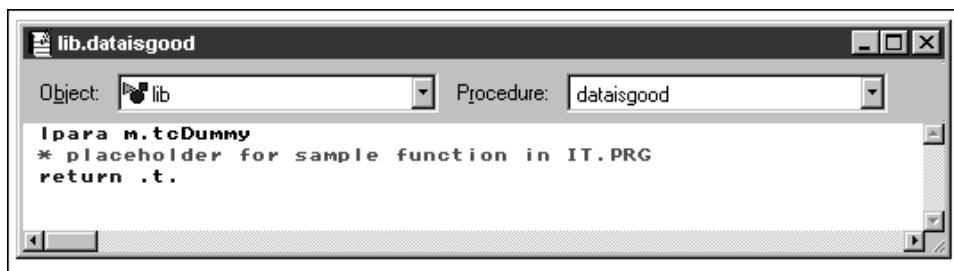
```
m.nAmtRAM = oLib.GetRAM()
```

Similarly, the call to the UserError routine, UserError, in MYPROC:

```
on error do UserError
```

would be replaced by:

```
on error oLib.UserError()
```



**Figure 15.7.** Creating a method in the LIB class to replace a similar function in MYPROC.

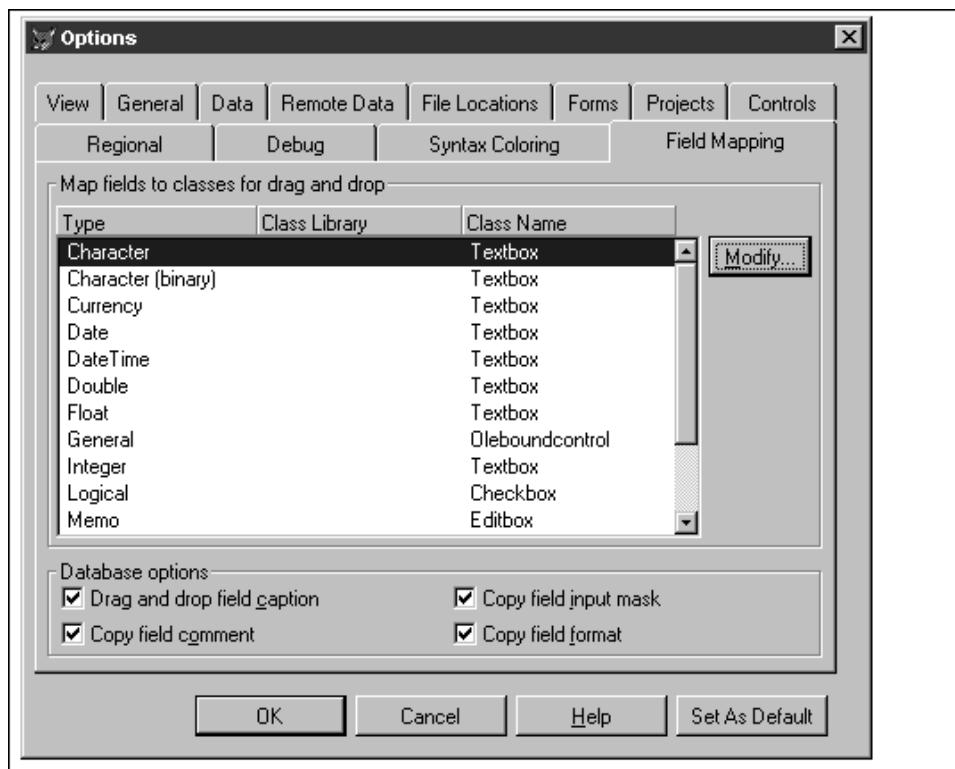
There is a danger in creating a class library to hold your routines: the increased cohesion between classes. In an ideal world, every object would be its own separate entity, and it could be plugged into anyone else's application, just like interchangeable parts. However, calls to other objects from within objects inextricably tie those two objects together, such that they end up having to always live together.

In a complex application framework, a certain amount of this is probably necessary, and the design and coding requirements to make the objects completely independent become onerous. Nonetheless, one shouldn't rush to create dozens of objects with a multitude of interdependencies. The efficiencies possible in object-oriented programming will disappear into the morass of impossible maintainability.

## Field mapping

Remember when you were dropping labels and text boxes onto a data-entry form instead of dragging them from the data environment? It was a bit of a nuisance because you had to manually set the control source of each control, but doing so enabled you to determine which class you used for each control.

As of version 5, Visual FoxPro allows you to set which of your own classes should be used as the base class for fields you drag onto a form from the data environment. This capability is provided through the Field Mapping tab of the Tools, Options dialog, as shown in **Figure 15.8**.



**Figure 15.8.** You can map fields to classes for drag-and-drop operations.

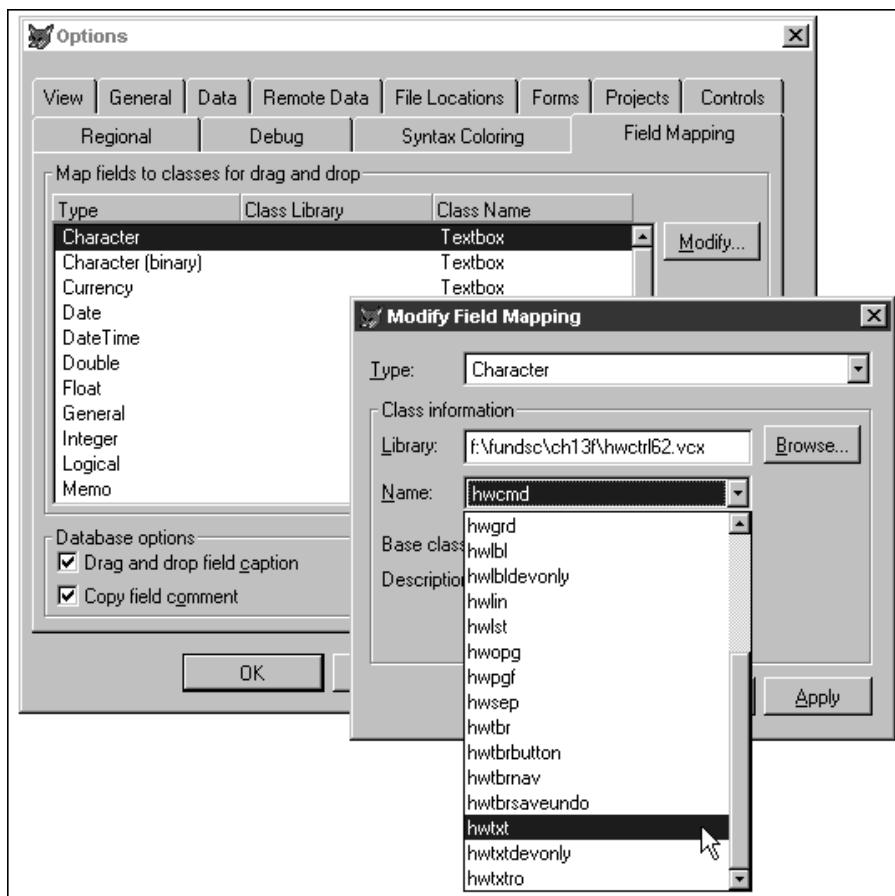
You can choose different classes (in different class libraries, even, if necessary) for each type of data. Select the data type in the Field Mapping list box, click the Modify button, and then select the library and class in that library, as shown in **Figure 15.9**.

Notice that you can set several options for each data type. You can choose to drag and drop the field name as a label for the control, and also copy the field comment, input mask, and format.

## Restoring the toolbar

Users are goofs, of course, and when they notice the little close box in the upper right corner of a toolbar's title bar, they're bound to try to click it. Then what? The problem is that opening another form does not bring the toolbar back—they have to close all the forms and then open them again. Not a killer, but perhaps it's one of those things that could be avoided.

There are several ways to deal with this circumstance. The first way would be to prohibit the user from closing the toolbar in the first place. Workable, but decidedly unfriendly. The second way would be to automatically activate the toolbar if the user opens a form and the toolbar isn't showing.



**Figure 15.9.** You can tie your own classes to fields for a variety of default operations.

A third way would be to provide a menu option to allow the user to call the toolbar back when they wanted to. I prefer this third method because it transfers control to the user. Suppose you'd chosen Door Number Two, and then imagine their frustration as they open four or five forms, and after each one, undock and then close the toolbar again and again. “I don’t want that toolbar up!”

I’m going to create a menu option to “Recall toolbar,” and the only question is where to put it. I suggest that it belongs under the Record menu pad so it’s unavailable when no application forms are active. Here’s how:

1. Add a separator menu option so that the Recall toolbar menu option isn’t jammed up against other menu options:
  - Change the prompt to “\-”.
  - Change the type to “Bar”.

2. Add a “Show toolbar” menu option:

- Change the prompt to “Show Toolbar”
- Change the type to “Procedure”
- Edit the procedure as follows:

```
if isnul(oApp.oToolbar)
    oApp.oToolbar = createobject(_screen.activeform.cToolbar)
    oApp.oToolbar.show()
else
    wait window nowait "Toolbar is already visible"
endif
```

3. Save and rebuild the menu.

When you open a form and then undock and close the toolbar, the Record, Show Toolbar menu option will activate the toolbar again. If the toolbar is already visible, you’ll get a message informing you so.

Here’s the logic. First, inside the procedure—when users close the toolbar while a form is open—the oApp.oToolbar reference changes to .NULL. You can see this for yourself if you put the following expression in the Debug window and then open a form:

```
oApp.oToolbar
```

The expression in the Debug window evaluates to “(Object)”. Close the toolbar (drag it off the top, and then click the close box). The expression now evaluates to .NULL.

Next, put the following expression in the Debug window and close the toolbar again:

```
isnull(oApp.oToolbar)
```

You’ll see this expression evaluate to .T., because oApp.oToolbar is null.

So the first line in the procedure checks to see if the object reference evaluates to null, and if so, brings it forward again. If the object reference exists, the toolbar is there and the user is so informed.

What about the case where no forms are open, and thus the toolbar won’t be around either? Easy! Because the user can’t get at the Record menu when there aren’t any forms open, this isn’t a worry. Of course, you could put a “nFormInstanceCount > 1 expression in the SKIP FOR dialog for the Show Toolbar menu option if you needed to put the Show Toolbar menu option elsewhere in the menu system.

Of course, you don’t have to do it this way. The ShowAppToolBar method of the App class contains the following line of code that determines whether or not to activate the toolbar:

```
if this.nFormInstanceCount = 0
```

To automatically activate the toolbar each time a form is opened and there is no active toolbar:

```
if this.nFormInstanceCount = 0 or isnul(oApp.oToolbar)
```

## Choosing between data sets

Because my company does a lot of credit-card charging and book shipping, we use a couple of applications that automate much of that work. Specifically, we use one application to authorize credit-card charges, and another to calculate shipping charges for UPS. Incredibly enough, both of these applications have a huge flaw: they both must be installed on your local C drive, and they bury their database deep in the bowels of the C drive as well. This means that you have to set up your backup procedure to dig a half-dozen interrelated files out of a directory buried six levels deep under “Program Files,” and woe be to you if your C drive crashes in the interim.

I’ve asked both companies why they don’t provide the ability to point to a data file located somewhere else, like, duh, on a server that is automatically backed up regularly, and the only answer was, “Well, uh, that would have been too hard.”

Don’t be like these losers—give the users of your application the ability to move their data set where they want it to be, and, even better, give them the ability to switch between data sets. This latter capability is just as important—many companies like to be able to switch between test and production data during testing, and to use the same application against multiple sets of data, such as for two companies run by the same people.

There’s a catch involved in switching data sets when using Visual FoxPro’s forms and data environments: The path to the DBC() is hard-coded in the read-only (at design time) Database property of each table in the data environment. If your data is in a different directory than your program, the full path is stored in the Database property. As a result, even if you use the SET PATH To or SET DEFAULT TO commands to change to a different data directory, your forms will still point to the original data set.

The solution I’ve come up with involves two steps. First, I always create a data directory that contains an empty copy of all databases and tables needed for the application, and I use that data set as the “pure original data set” for all time. Whenever I need to make changes to the data structures, I make changes to that data set, and then propagate the changes to other data sets as needed.

Second, I place the code along the lines of the following in the BeforeOpenTables method of every data environment (kudos to my tech editor, Doug Hennig, for creating the wrapper that grabbed every cursor object in the data environment):

```
*  
* the following code snippet goes into the BeforeOpenTables method of every DE  
*  
m.lnEnv = amembers(laEnv, this, 2)  
for m.lnI = 1 to m.lnEnv  
  oObject = eval("this." + laEnv(m.lnI))  
  if upper(oObject.baseclass) = 'CURSOR'  
    m.lcDBC = oObject.Database  
    oObject.Database = fullpath(oApp.cLocCurData + substr(m.lcDBC, rat("\\",  
m.lcDBC)), curd())  
    endif && upper...  
  next m.lnI  
oApp.lCurrentFormHasMultiDataSetsEnabled = .t.
```

This could be fairly obscure if you’re not used to reading code with object references in it, so I’ll walk through each line.

The first line determines how many objects are in the data environment (notice the subtle “this” second parameter that refers to the data environment (remember, this code is in the DE!). Then, the code loops through for each object in the DE. If the object is a cursor, the current database location (`oApp.cLocCurData`) is stuffed into the cursor’s Database property.

The last line sets the `oApp.lCurrentFormHasMultDataSetsEnabled` property to True. This is necessary because this chunk of code must manually be put into each data environment—since the DE can’t be subclassed. The global application object, `oApp`, has this property that is by default set to False. Each data-related form class checks this property, warns the user (hopefully, the developer) that multiple data sets were not enabled, and then shuts down the form.

If this is not done, the user could possibly use the form, not aware that the form is actually pointing to a different data set. It’s nearly impossible to find bugs caused by incompatible data that’s coming from two different databases in the same application.

## **Developer-only controls**

Remember that pair of controls in the Order Maintenance child form? A label and a text box were different colors—if you opened the form in the source code for that section, you saw that they were both yellow, and that they were created from different classes than the other labels and text boxes on the form.

I’m a dope most of the time, and I’ve found it necessary to resort to all sorts of tricks to help me along in the day-to-day business of writing apps. One crutch I use over and over is to display data on forms that is of interest to me but that wouldn’t be of interest (and, most often, would end up being confusing) to the user. One example is the primary key of a table in a form. I always include a text box on a form that displays the PK for the table, as well as any foreign keys that are also in that table. Having that information is always handy when trying to chase down bugs. Of course, you wouldn’t ever want your users to see that data.

As a result, I’ve created two special classes—one for labels and one for text boxes—that I use to label and display data like this. The classes have yellow backgrounds so I can tell immediately that these fields are for my eyes only. The only trick is this: How do I get these objects to show up when I’m running the app, but not when my users are running the app?

The answer has two parts. First, I always create a “developer-only” flag in my apps that I can turn on and off. In the olden days, I used to use DOS SET commands, but those are more and more inconvenient to use with Windows. Now, I resort to a quick and dirty method: In my application object, I look for a file named “`FPDEV.WHIL`” in the root directory of the drive where my application is running. If the file is there, then I’m in “developer mode”; if it’s not, the app is being run by an ordinary user.

The existence of this file sets the “`oApp.cMethod`” property to “`DEV`”; the absence sets the property to “`USER`”. Then, I can simply test for

```
oApp.cMethod = "DEV"
```

and take the appropriate action. Here’s the code in the `Init()` method of the APP class:

```
App.Init()
if file("\FPDEV.WHIL")
  this.cMethod = "DEV"
```

```
else
  this.cMethod = "USER"
endif
```

Now back to the label and text box that disappear when a user is running the app. I created two classes, hwlblDevOnly and hwtxtDevonly, both based on VFP base classes, and put the following code in their Init() methods:

```
* hwlblDevOnly.Init()
* hwtxtDevOnly.Init()
if oApp.cMethod = "DEV"
  this.backcolor = 65535 && 8454143 = light yellow, 8454016 = light green
else
  this.visible = .f.
endif
```

Now, when a form with either (or both!) of these controls runs, the control is either displayed or made invisible according to the value of the oApp.cMethod property. I get my data the way I want it without annoying or confusing my users.

Of course, you can use oApp.cMethod to drive a whole set of behaviors for your application. The specifics are left to you, the reader, as an exercise.



## **Section IV**

# **The Development Environment**

The basic idea behind computer programming is that of automating the tedious and redundant processes that humans used to have to do manually. It only makes sense that you should spend a bit of time automating your own processes as well, doesn't it? In this section, I'll go over a number of disparate but related topics that will make you more efficient and a better developer.



# Chapter 16

## Customizing Your Development Environment

**It never ceases to amaze me to walk into a development shop and see how the developers work. The most mind-numbing experience is to watch a developer start up Visual FoxPro, and then issue three or four attempts of the “CD <current app dev directory>” command as his first step—because VFP’s default location is still set somewhere deep in the bowels of Program Files. Think of all the time you spend building applications to make other people more productive. Doesn’t it make sense to tune your own work environment? In this chapter, I’ll show you a number of techniques that you can use to make your daily work easier and more bug-proof.**

The purpose of this chapter is to introduce to you some ideas and concepts on how to tune your development environment. But before I start, let me warn you that the methodology presented here is just one means to an end—one way of accomplishing three important goals.

The first goal is to customize the environment so that it works well for you and your style. Different developers have different needs—one of you is probably going to spend the next two years on a single app, while another will work on pieces of over a dozen applications. Some of you are going to stay strictly LAN-based for the foreseeable future; others are heading full barrel into Web-based and distributed development. No one style is going to work the same for everyone.

The second goal is to optimize your development environment. Do you really want to go through four steps every 10 minutes when you could simply tap a hot spot on your computer screen or press a hotkey? Just like “Quick Carl” in those old Marathon Bar advertisements, I like to do everything “fast-fast-fast”—and I hate waiting for anything on my computer as well. Ever get up and walk into another room to get something, and by the time you’re in that room, you’ve forgotten what it was you were going to get? Application development is based on ideas, and as your brain is cranking out the idea, the trick is to keep the response time or turnaround time of the machine shorter than your attention span.

The third, and most subtle, goal is to set up shop so that you are less likely to create bugs. Some of those attributes have nothing to do with the bits and bytes on your hard disk—a big work desk, a big monitor, plenty of hard disk space, an office where you can close the door and shut out the noise so you can concentrate—they’re all part of the package. But there are lots of things you can do while configuring Visual FoxPro and structuring your application development directory to prevent you from doing “stupid” things. I’ve long avoided keeping an open beverage on my desk; I feel an open mug of liquid perched above my system unit, ready and waiting for me to knock it over, is an accident waiting to happen, and I think I’ll try to avoid that one. Similarly, how can you construct your development environment so as to avoid those “accidents waiting to happen”?

All too often I see directories for all of a developer’s custom apps located under the Visual FoxPro directory, right next to “\API”, “\DISTRIB.SRC” and “\FFC.” Or if they’ve made the

conceptual leap of making separate directory entries from each application, perhaps even all stored in a parent “applications” directory, the entire application is stored in one directory—source code, data, common libraries, third-party utilities, and so on. It’s a recipe for disaster if there ever was one. It’s all too easy, for instance, to take a copy of the application to the customer’s site and accidentally overwrite their live data with your copy of the table that contains entries for Bugs Bunny and Jeri Ryan. I imagine it would be pretty embarrassing to have to hunt down the network administrator to find out how recent their last backup is after blowing away 170 megabytes of data.

You’re certainly free to pick and choose from the ideas I’ll present here and determine what works best for you. I’ll present problems I’ve run into in the past, mention some problems that you’ll likely run into in the future, and present some strategies and techniques for dealing with those issues. And I’d love to hear from you about tricks and tips you’ve developed to tune your own environment.

To get started, then, I’m going to begin at the beginning. I brushed over installation in Chapter 1 because you needed to get into the meat of VFP right away. Now that you’re comfortable with data, programs, and classes, and have even possibly built a couple sample applications, it’s time to set up things correctly.

## **Machine setup and installation**

I think it’s a reasonable assumption that, at the end of the 20th century, every database developer on the planet should be developing on a LAN, or at least have access to one on a regular basis. However, I also think there are significant benefits to doing your primary development on a workstation. As a result, I’m going to be a little schizophrenic in my suggestions.

First of all, you have to install Visual FoxPro on your local workstation. Unlike older programs, where you could get away with an install on the server and then run the program from that box, current Windows programs are tightly integrated with the workstation upon which they’re running. (This also goes for applications you build for others—they have to install your application on each user’s machine.)

Before you go any further in the installation process, consider this: What should your computer look like? I’m going to assume that you keep all of your “data” on a server of some sort; if you don’t, you can just substitute a drive on your local machine for the server. I think it makes sense to have at least two drives available on your local machine. One will be for your shrink-wrap software, and the other is for your data.

Figure you’re going to have to reinstall Windows sooner or later (probably sooner)—don’t be one of those sorry goofs who’s got 800 megabytes of data buried in the depths of their various C-drive directories. I spent a couple of hours on the phone with a frantic acquaintance after she had watched her machine disintegrate before her eyes during the Office 2000 beta. Turns out she had just taken the computer out of the box and started working with it. One C drive—that’s all. Had installed beta after beta on the machine over a period of years, trusting that “uninstall” actually would. She had literally thousands of documents stored in My Documents, and thousands more data files scattered throughout the rest of her drive. It took her a week to back up those files to a hastily purchased ZIP drive before she could get her machine working again.

---

As you pull out the two or three dozen CDs (well, it seems like that many! Remember when software used to come on a single CD?) that make up Visual Studio, you'll realize that the 2-gigabyte partition that your C drive was created in no longer seems sufficient.

You have two choices. If you have the luxury of creating a larger C drive partition, 4 to 6 gigabytes should be enough for most developers. (I still remember being able to boot and run my primary development environment from a 1.2-megabyte floppy!)

If you're stuck with a 2-gigabyte C drive, as I am (I got a new box a few months ago and I just couldn't bear to do another FDISK and NT reinstall this year), the alternative is to do minimal installs of everything you possibly can. I do a complete install of VFP and VB, bail on the installs of other Visual Studio programs (C++ takes 500 meg, and I'm not about to become a C++ junkie), and then selectively pick and choose among options in Office and other programs. In addition, I've located MSDN on my 2-gigabyte D drive—allowing me to load every help file I could find on the drive with some room to spare.

In either case, the primary philosophy I recommend is to keep everything except shrink-wrap off of your C drive. There are two kinds of developers—those who can FDISK and reinstall NT without turning the monitor on, and those who are going to learn to.

As a result, I have a binder with step-by-step instructions on how to set up each machine in my office: the equipment inside the box (for example, weird video cards), the programs that had to be loaded, the location of the programs (do you have 1700 CDs lying around your office, too?), and the various settings (such as IP addresses) needed at each point during the install.

Because each box that shows up ends up being configured a little differently (ZIP drives, CD-ROM or DVD drives, video cards, adapters, and so on), I also have a directory on my server that contains a copy of every driver and program that needs to be installed on each machine. That way I can keep straight the Diamond drivers for my Tahoe machine and the STB drivers for my Indy machine. (I also keep a copy of these directories on the local drives just in case the install—or reinstall—hoses the network connection.)

I'll address the rest of your setup—directories for data and whatnot—shortly.

## Startup configuration

The first time you click on the Visual FoxPro icon, you'll be presented with the Welcome screen. Just about every developer I know immediately checks the "Don't ever, ever, ever show this screen again" check box in the lower left corner of the form. If you're one of those people but want to run the form again, you can launch the VFP6STRT.APP program in VFP's home directory.

By default, VFP 6.0 is installed the C:\Program Files\Microsoft Visual Studio\VFP98 directory.

## Default directory

As I mentioned earlier, you don't ever, ever want to start Fox in this directory, because sooner or later you'll end up creating files in that directory. You can control the VFP startup location through two different mechanisms.

The first is to simply change the "Start in" location in the desktop icon's Properties dialog. (Right-click on the icon on the desktop and select the Properties menu command.) Unfortunately, there's no "browse" button, so you have to manually enter the location yourself.

By the way, if you've enabled the Windows Quick Launch toolbar in the Start menu task bar, you can drag a copy of your VFP desktop icon to it. (Right-click in the task bar, and select the Toolbars, Quick Launch menu option to enable it.) I do this for the half-dozen or so applications that I use all the time, instead of having to close or minimize all open windows to get to my desktop icons. However, once you copy the desktop icon to the Quick Launch toolbar, the values in the Properties window take on a life of their own. Specifically, if you change the "Start in" property in the desktop icon's Properties window, that change is not reflected in the Quick Launch icon's Properties window. This drove me nuts for a couple of days until I realized I was launching VFP from both icons at different times.



By the way, the default startup location isn't the only thing you might want to change. If you've got more than one version of Fox (any version) installed on your machine, you might find it confusing to have multiple icons on your desktop. That's why I've kept creating new icons for each new release of FoxPro since FoxPro 2.0 was released sometime in the early 17th century. You can find icons for each version since 2.6 in the CH16 source code downloads for this book.

Okay, enough whimsy. You can also change VFP's default directory in the File Locations tab of the Tools, Options dialog, as shown in **Figure 16.1**.

Entering a value in the Tools, Options dialog will override any value you enter in the Properties window of the VFP desktop or Quick Launch icon.

However, just because you can now automatically load VFP and have it start in a different default location doesn't mean that you'll never return to VFP's home directory. There are lots and lots of goodies in the various subdirectories—a rich variety of samples and useful programs, as well as the wonderfully robust Fox Foundation Classes (FFC). So it's entirely possible that you might quickly get sick and tired of trying to change back to this directory by navigating through the various Open dialogs in Windows. If you're in VFP and you've changed VFP's default directory location, you can switch back to VFP's home on C with the HOME() function. For example, typing:

```
cd home()
```

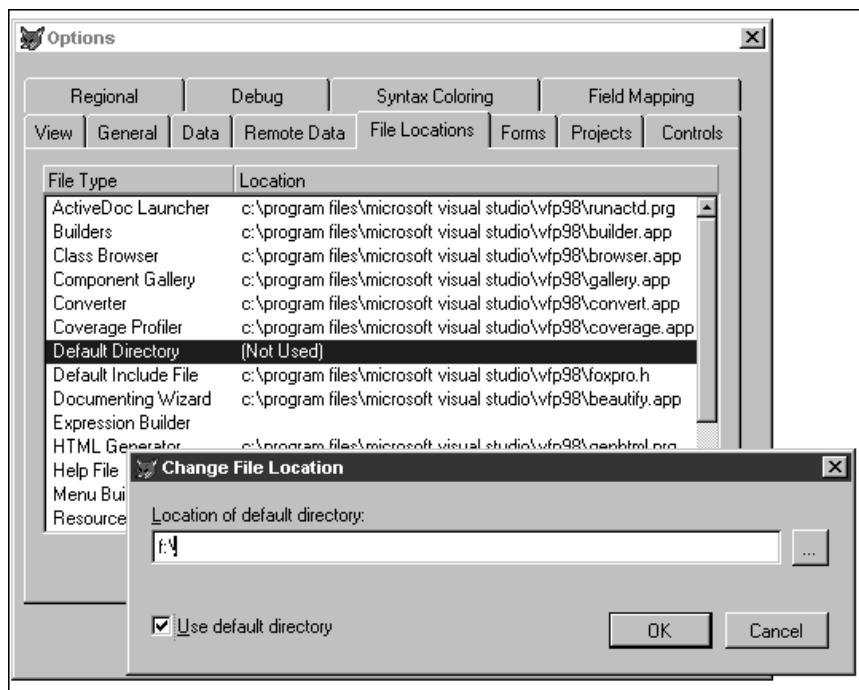
in the Command window will perform the equivalent of:

```
cd "C:\Program Files\Microsoft Visual Studio\VFP98"
```

or even:

```
set default to "C:\Program Files\Microsoft Visual Studio\VFP98"
```

Remember to enclose these fully qualified paths in quotes, because they contain spaces.



**Figure 16.1.** To change Visual FoxPro's default directory, check the Use default directory check box and then enter or choose a directory.

## Startup files

When Visual FoxPro starts, it looks in a variety of places for information on how to set itself up. About a thousand places, I think. You just saw one attribute of VFP's setup data—the location of the default directory. The files of interest are the Windows Registry, CONFIG.FPW, FoxUser.DBF, and FoxPro.INI. Here's the deal on each of them.

### The Windows Registry

In the olden days of Windows 3.1, configuration data was stored in text files that had the extension of INI (for “initialization”). There were three problems with these files: there were lots of them, they could be edited by civilians, and it was difficult and/or impossible to manage them remotely. Thus, Microsoft came out with a binary data store called The Registry. It provides a single location for all Windows-related configuration data, and requires a special tool, RegEdit, to get into it. It can also be accessed remotely, if you really know what you're doing. All settings in the Tools, Options dialog are stored in the Windows Registry.

## **CONFIG.FPW**

This text file stores the settings of a number of Visual FoxPro SET commands as well as several other options that can't be controlled from anywhere else. The default CONFIG.FPW file is located (where else?) in the Visual FoxPro home directory.

However, you can tell VFP to use a CONFIG.FPW file located elsewhere. The answer, however, is not that straightforward. Because of the variety of environments that developers work in, and the types of applications Fox developers have built over the years, there are a considerable number of places where you can do so, and there is a sequence of steps that VFP follows to determine which CONFIG.FPW file to use.

First, you can use a command-line switch (in the Target property of a desktop shortcut) to specify a particular config file. In fact, you don't even have to name it CONFIG.FPW! See the following section on startup switches for an example or two.

Next, VFP checks the contents of the DOS environment variable FOXPROWCFG (remember DOS?) for the fully qualified path of a file. Again, you don't have to name the config file CONFIG.FPW. If neither of these two choices applies, Visual FoxPro checks the default directory and then the DOS path (if one has been established) for a file specifically named CONFIG.FPW.

You can determine the current config file in use with the SYS(2019) command. If no config file was used, this function returns an empty string.

When you ship an application with an .APP or .EXE file and the VFP runtime, the config file business requires an additional step. You can include your own config file in your app (you have to actually bind it into the app); doing so overrides all of the other steps. If you don't include a config file, VFP follows the normal route of searching for the config file.

## **FoxUser.DBF**

There is a .DBF (and associated .FPT) called FoxUser that can be used to store preferences, such as the positions and sizes of all windows in your development environment and preferences for editing windows. Each time VFP closes down normally, the last settings of those preferences are saved to individual records in this .DBF. That's why, if you move your Command window around, it shows up in the same place when you fire VFP back up.

This .DBF is called the resource file, and you can turn it on or off in a couple of ways. First, you can issue the commands SET RESOURCE ON/OFF in the Command window. Like the config file, FOXUSER is normally stored in the VFP home directory but can be located in a different directory. Again, like config, there are several mechanisms to determine which resource file to use. You can use the Resource file entry in the File Locations tab of the Tools, Options dialog to store the location in the Windows Registry. You can also put an entry in your config file, like so:

```
RESOURCE = <name of file>
```

Doing so overrides the setting in the Registry. And last and least, you can SET RESOURCE TO <name of file> interactively.

## **DEFAULT.FKY**

You can create keyboard macros to automate the execution of special tasks. These macros can be stored in a file of your own choosing, but Visual FoxPro will automatically load any keyboard macros found in a file called DEFAULT.FKY located in the VFP home directory.

## **FoxPro.INI**

FoxPro for Windows and Visual FoxPro 3.0 both used a file called FoxPro.INI to store settings such as the various startup window positions. These settings are now stored in the Registry. This .INI file was located in the Windows directory. In my brief testing I couldn't tell if it still works, but even if it did, you shouldn't use it. Try to keep everything in as few places as possible—which means the Registry in this case.

## **How to use the startup files**

So now that you know what they are, how should you go about using them? First of all, you can't do much about the stuff that's stored in the Registry.

Remember my rule about keeping as little info on drive C as possible? Well, you can't move the registry off of C (and if you figured a way to do so, it would probably break Windows six ways from Sunday because of its tight coupling with the operating system). That's why I don't keep a config file, a resource file, or anything else on my C drive anymore. Nor do I use DOS environment variables, because by definition they're "part" of C as well.

Instead, I make a few small changes to VFP's File Locations in Tools, Options, and point to files on another drive. Then I configure the daylights out of those files. If I have to rebuild a machine (or a new version of Fox comes along, possibly ignoring or wiping out all of the files in my previous VFP home directory), I just point to the files on the other drive, and I'm all set.

I keep all of these files (as well as a host of others) in a directory called "DeveloperUtilities" on my main VFP development drive. Actually, there are several of these directories, one for each version of Fox that I'm using, because utilities for one version often won't work in another.

Here are the settings I change: Default Directory, Menu Builder, Resource File, Search Path, and Startup Program.

### **Default Directory**

I've already addressed this, but to be complete I wanted to make it clear that I change the default directory here, not in the Properties dialog of the shortcut.

### **Menu Builder**

This points to Andrew Ross MacNeill's GENMENUX program that I discussed in Chapter 7, "Building Menus." This program is stored in my Developer Utilities directory.

### **Resource File**

I keep my FoxUser file in my Developer Utilities directory as well. Remember that FoxUser consists of both .DBF and .FPT files.

## Search Path

The VFP search path is much like the old DOS search path—it's where VFP will look for files if they're not in the current directory. This includes programs, forms, and even databases and tables! This is particularly handy because, after all, what good are a bunch of cool utilities in your Developer Utilities directory if you can't get to them easily? I keep my Developer Utilities directory in my search path at all times.

## Startup Program

Here's where all the real work happens. You can have Visual FoxPro automatically run a VFP program when it starts. That's what the Welcome to Visual FoxPro splash screen really is—just a VFP program.

But before I get into the details of what your startup program might do, I should mention that, as with all things FoxPro, there are a bunch of ways to specify the startup program. This entry in the Tools, Options dialog is one, of course, but you can also make one of two entries in your config file:

```
_STARTUP = <program name>
COMMAND= <command>
```

For example:

```
command = do \MYPROG3.PRG
_startup = "\MYPROG2.PRG"
```

Note that you need to specify a complete command with COMMAND—not just the name of a program to execute. If you specify programs in both, both will run: the program attached to \_STARTUP first, and then the one attached to COMMAND. The program assigned to \_STARTUP in your config file updates the value you enter in the Tools, Options dialog.

I personally just use the entry in Tools, Options, and point to my startup program in my Developer Utilities directory like the rest of the settings I've mentioned in this section. So what could you do in a startup program?

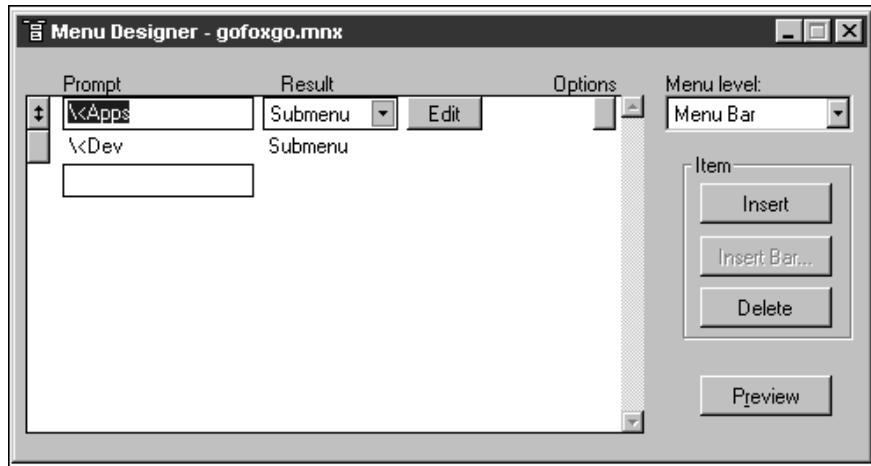
I use mine to launch a couple of applications that I use all the time, add a couple of menu pads to the standard VFP menu, open a couple of windows, and then place focus on the Command window when I'm all done. Here's the code:

```
* gofoxgo.prg
* start with a good menu
set sysmenu to default
* open the class browser
do (_browser)
* (other programs you want to load can go here too)
* run my custom menu
do GOFOXGO.MPR
* change the title of the VFP window to show what version
* of VFP I'm running as well as the current drive and directory
modify window screen title "We're developing in " ;
    + version() + " (" + sys(5) + sys(2003) + ")"
* open up the Data Session window
set
```

```
* place focus back to the Command window
keyboard "{CTRL+F2}"
* get rid of anything that was left laying around on the VFP desktop
clear
```

My tech editor has made a number of other suggestions as well. First, SET ASSERTS ON should be the default during development. This is needed because there isn't a Tools, Options entry for this setting. SYS(3050, 1, 24000000) sets the maximum amount of RAM that VFP can use to 24 MB. Otherwise VFP will try to chew up all the memory on the machine, which can then cause problems if you try to open many other apps at the same time. And SET SYSMENU SAVE will save all these settings so that issuing a SET SYSMENU TO DEFAULT command won't blow it all away.

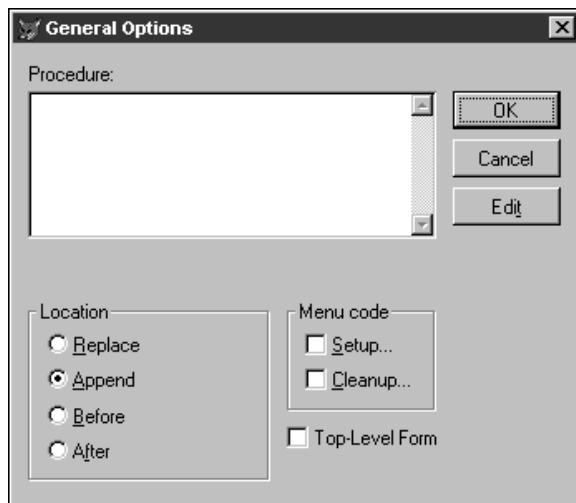
As you can see, this code is located in a program named GOFOXGO, and this .PRG is located, of course, in the Developer Utilities directory. The GOFOXGO menu, referenced about halfway through the program, actually adds two menu pads to the VFP system menu instead of replacing it. The pads are shown in **Figure 16.2**.



**Figure 16.2.** The GOFOXGO menu adds an “Apps” and a “Dev” menu to the VFP system menu.

The Apps menu pad displays a list of applications I'm currently working on. Selecting any of the applications changes the current directory to the directory that contains the source code for that project. The Dev menu pad displays a variety of developer utilities and helpful shortcuts.

To add these two pads to the existing menu, select the View, General Options menu command, and select the Append option button, as shown in **Figure 16.3**. If you don't, you'll end up with a menu that has only two menu pads, and the rest of the Visual FoxPro menu will have disappeared.



**Figure 16.3.** Be sure to select the Append option button to include additional menu pads on the standard VFP menu.

In order to keep the code for the menu as stable as possible, each menu command runs the same program, passing the name of the menu bar as a parameter:

```
do l_gofox in GOFOXGO with prompt()
```

The L\_GOFOX routine, contained as a procedure in GOFOXGO.PRG, looks like this, in part:

```
*****
function l_gofox
*****
lparameter m.tcNaApp
m.cPathPrefix = "f:"
m.cPath60LAN ;
    = m.cPathPrefix + "\devuti60; " + m.cPathPrefix + "\common60LAN"
m.cPath60CS ;
    = m.cPathPrefix + "\devuti60; " + m.cPathPrefix + "\common60CS"

do case
case upper(m.tcNaApp) = "BC"
    set path to &cPath60LAN
    set default to m.cPathPrefix + "\bc\t12\source"
case upper(m.tcNaCust) = "WEAC"
    set path to &cPath60CS
    set default to m.cPathPrefix + "\weac\mid2\source"
```

```
case upper(m.tcNaCust) = "WHERE AM I?"
    modify window screen title "We're developing in " ;
        + version() + " (" + sys(5) + sys(2003) + ")"
case upper(m.tcNaCust) = "DUPLICATE CURRENT RECORD"
    scatter memo memvar
    append blank
    gather memo memvar
case upper(m.tcNaCust) = "DEVHELP"
    do DEVHELP
case upper(m.tcNaCust) = "SWITCH TO DEVUTILS DIRECTORY"
    set path to m.cPathPrefix + "\devuti60; " + m.cPathPrefix + "\common60"
    set default to m.cPathPrefix + "\devuti60"
other
    wait wind "You goofball! Better clean up the menu!"
endcase

* now change the window title to reflect our current location
modify window screen title "We're developing in " ;
    + version() + " (" + sys(5) + sys(2003) + ")"
clear
clear program
return .t.
```

Because I might be using different class libraries for different projects, the first thing that the L\_GOFOX function does is create extended paths for the locations of the class libraries. COMMON60LAN contains one set of class libraries and is used for one group of applications, while COMMON60CS contains a different set of class libraries and is used, obviously, for a different set of apps.

Then the program processes each menu option passed to it in a large CASE statement. Most of these CASE segments will just set the path and change the default directory, but a few will perform different kinds of tasks. For example, under the Dev menu pad, I have a command called “Duplicate current record”—and all it does is scatter memory variables from the current record, add a new record, and then gather the recently scattered memvars into the new record. Simple, but it sure saves a lot of time. Another menu command under the Dev menu pad automatically switches to my Developer Utilities directory.

After any of these routines is run, I’ll update the window title. It sounds silly, but I’ve found it really useful to see the name of the current directory in the title at all times. As soon as the phone rings, my concentration is shot.

### A word about the help file

Another file location that many people like to alter is the Help file. If you’ve got the *Hacker’s Guide to VFP* (which you should—it can be purchased online at [www.hentzenwerke.com](http://www.hentzenwerke.com)), you can copy the HACKFOX.CHM file to, say, the Developer Utilities directory on your machine, and then point to it instead of to FoxHelp. I actually recommend not doing so, however. Instead, point to MSDN help in the Help File location in Tools, Options, and then either place shortcuts on your desktop/QuickLaunch toolbar, or add menu options to the native Fox menu to point to the Hacker’s Guide .CHM (as well as any other .CHM files you want to regularly access). The reason is that you’ll often find yourself needing information that is not Fox-specific. If you call up MSDN help inside Fox, you can access anything—not just Fox.

If you've gone ahead and changed the Help File location and now want to change it back, you might get a little frustrated trying to figure out where to change it back to. First of all, help files are no longer stored in the Visual FoxPro home directory; instead, everything is stored in the MSDN directory. Typically, that's under the Program Files\Microsoft Visual Studio directory (although it may vary if you've installed MSDN somewhere else). Second, "help" actually consists of several hundred .CHM (and related) files deep inside the MSDN directory. Which file is the "main" file? It's not even a .CHM file—it's a custom viewer that Microsoft uses to provide additional functionality, like the menu at the top of the MSDN help window. The name of the file is MSDNVS6A.COL, and it's (as of this writing) located with the rest of the .CHM files—in the subdirectory MSDN98\98VSA\1033.

You might be thinking that you could create a shortcut to the MSDNVS6A.COL file, and keep VFP pointed to HACKFOX.CHM, but, unfortunately, shortcuts to MSDNVS6A.COL generate an error.

## **Startup switches**

You can include "command-line switches"—additional characters after the name of the program in a shortcut's Properties dialog—to control how Visual FoxPro will load.

Use the -A command-line switch to start FoxPro without reading the current CONFIG.FPW and Registry switches. This is handy if one of those files (particularly the Registry!) has become corrupt, and if you can't load VFP any other way. Example:

```
C:\Program Files\Microsoft Visual Studio\VFP98\VFP6.EXE -A
```

Use -R to re-establish the associations that Visual FoxPro has with file extensions in Explorer and File Manager. This is a lot easier than having to edit each file association manually:

```
C:\Program Files\Microsoft Visual Studio\VFP98\VFP6.EXE -R
```

Use the -C switch to control which config file VFP will use during startup:

```
C:\Program Files\Microsoft Visual Studio\VFP98\VFP6.EXE -CF:\MYCONFIG.FPW
```

If you find yourself using any of these switches frequently, you might consider creating multiple VFP startup icons on the desktop, each with its own VFP configuration and startup switches.

## **Developer vs. user requirements**

Your needs as a developer and the needs of your users while they are using your application are considerably different. You'll often find yourself needing to perform functions for testing and debugging—while your application is running—that you don't want your users to have access to. For this reason, it's a good idea to have a separate menu pad in your application that provides access to a variety of "developer-only" commands.

These developer-only commands might include those shown in **Figure 16.4**.



**Figure 16.4.** A typical developer-only drop-down menu.

The first five commands allow the developer to open any of the debugging windows. The next three are windows I find handy—the View window (now known as the Data Session window) is particularly useful while an app is running because it allows you to investigate what's happening in each of the open data sessions. The next batch simply lets you halt and continue execution of your program. And the last one opens up the Event Log—the file that all VFP errors are saved to.

Of course, what you put on your Developer menu is limited only by your imagination. For example, my tech editor also adds these menu options:

- Object under Mouse (F12). Performs “o=sys(1270)”. This creates an object reference, called “o”, to whatever is under the mouse when F12 is pressed. You can bring up the Watch window, enter “o”, and then view or edit properties of the object through the Watch window.
- Release O—Performs “release o”. This just releases the reference created in Object under Mouse (F12).
- Toggle Application Timer—Performs “oApp.oAppTimer.Enabled = !oApp.oAppTimer.Enabled”. Doug uses an application-level timer to refresh the application toolbar, but it's a nuisance when tracing code, so this allows the timer to be turned off and back on again during tracing.

I also add an extra menu command to the bottom of the Forms menu. Users see only one “Exit to Windows” command, while developers can choose “Exit to VFP” as well. In Chapter 15 I also described a text box class that displays on a form only when a developer is using it.

Furthermore, you will need the ability to run your application in a couple of modes—not only as a developer, but also as a user, to mimic exactly what users will see. Not only will they

get different menu options, but their screens will look different (because they're not seeing any debugging information you've included), and you might even want to run against a different data set.

It is often necessary to be able to run the application you are creating as if you were a user. Unfortunately, it's not very convenient to build a brand new executable and copy that file onto a machine that has only the VFP runtime on it.

And it's not unlikely that you'll also be called to a customer's site and asked to debug a running application—it's nice to be able to "set developer on" while you're running the application but not let your users have that same functionality.

Finally, I've found it useful to provide secret "back doors" into functions for specific individuals, such as myself, regardless of who has logged on to the application.

The requirements, then, are such:

- Ability to run as a developer or a customer
- Ability to simulate the VFP development environment or the customer's environment
- Ability to run as a specific user, regardless of logon capabilities.

I've found the easiest way to do this is to create three files in the root directory of the drive that the application is running on, and then do a quick test for those files in the startup program of the application. The existence of those files sets application object properties that the rest of the application can read to determine behavior. For example, here's a code segment that you could use at the beginning of your startup program:

```
lparameters m.gcMethod, m.gcLocation, m.gcByWho
do case
case pcount() < 1
  m.gcByWho = iif( file("\fpdev.whil"), "WHIL", "NONE")
  m.gcLocation = iif( file("\fplocation.hw"), "HW", "NONE")
  m.gcMethod = iif( file("\fpmethod.dev"), "DEV", "NONE")
case pcount() < 2
  m.gcByWho = iif( file("\fpdev.whil"), "WHIL", "NONE")
  m.gcLocation = iif( file("\fplocation.hw"), "HW", "NONE")
case pcount() < 3
  m.gcByWho = iif( file("\fpdev.whil"), "WHIL", "NONE")
endcase
```

You'll notice that this code allows the person running the program to specify these values in one of two ways: either by providing fields named a certain way in the current drive's root, or by passing parameters to the main program. In either case, default values (which mimic a customer's environment) are provided in case no parameters were passed or no files existed.

Each of these memory variables—m.gcMethod, m.gcLocation, and m.gcByWho—is converted into an application object property with the following code as soon as the application object is instantiated:

```
oApp.cMethod = m.gcMethod
oApp.cLocation = m.gcLocation
oApp.cByWho = m.gcByWho
release m.gcMethod, m.gcLocation, m.gcByWho
```

---

The Method parameter can be either “DEV” or “CUST”. If the parameter is “DEV”, the Developer menu pad appears, the “Exit to VFP” menu command appears, and all developer debugging controls appear.

The Location parameter can be either “HW” or “CUST”. This value is used to configure the environment’s data directories and other environmental options. The purpose is to allow the developer to simulate, as close as possible, the customer’s environment. I’ve not run across a situation where I’ve set this option to “HW” at a customer’s site.

The ByWho parameter is generally used for special cases. Typically, the developer would include a special branch of code that tested for this variable being equal to a value, like so:

```
if oApp.cByWho = "FPDEV.HERMAN"
  <code that only Herman wants to run>
endif
```

The nice thing about this one is that it’s easy to expand its functionality to multiple developers, simply by using additional extensions for the FPDEV file name.

Finally, I should note that the contents of these files, FPMETHOD, FPLOCATION, and FPDEV, are irrelevant—they merely represent the existence of the file itself. This means you can quickly and easily create these files anywhere, simply by using a text editor or copying another file and renaming it.

## Configuring development application directory structures

### A little bit of history, or “how we got here”

When I started developing database applications, “state of the art” meant you had a second 360K disk drive and 256K of RAM. As a result, you were forced to keep your programs and data in the same location. DOS didn’t even know about directories, much less hard disks! Of course, the data sets being handled at this time weren’t that big—and a key component of any specification was the delineation of the record size and expected number of records.

Human beings being subject to inertia like anything else, when that monster 10 MB hard disk came along, developers generally didn’t see any reason to change the tried-and-true method of placing data and source code in the same directory. In fact, there was a good reason for not doing so: a lot of code would have to be rewritten to handle this situation, and who was going to pay for that? Pretty hard to explain to a customer that they were going to have to shell out extra bucks for no perceivable increase in functionality.

Suddenly—overnight, it seemed—applications were running on 80386 processor-based computers and, with the speed capabilities of FoxPro, data sets of 25, 50, 100 MB or more of data were within the grasp of the developer!

This situation started to present a couple of problems. First was the issue of accidentally overwriting customer data when you delivered an update to a system. Sure, you could be real careful when updating the files, taking care not to stomp on any .DBFs. But you only had to goof up once. Or you could just take the files that had been changed since the last update. But that meant keeping track of update dates, and even then, it was possible to have dozens or hundreds of files that had changed in a large system. And if you were continually bringing out

incremental changes, how did you deal with the issue of discarded and outdated files? You didn't dare erase all the files, so when REPORT5.PRG was replaced by REPORT5A.PRG, you just brought REPORT5A.PRG along and left REPORT5.PRG to gather dust.

As I'm writing this book, my company inherited an application that has more than 600 files in one directory. (And we found out that they had just moved this application from the root directory into its own subdirectory not too long ago.) The first job is to clean out the junk, and it looks like we'll get down to fewer than 100 files by the time we're done. Imagine the disk space savings, not to mention the boost in productivity when doing additional maintenance on the system!

With the advent of FoxPro 2.0, the number of source code files required for an application nearly doubled. Instead of having a single .PRG (and its compiled .FXP) for a screen, two additional files were needed—the .SCX and the .SCT—and then two more: the generated screen .SPR (and its compiled .SPX). Menus presented the same situation with the .MNX/.MNT/.MPR/.MPX combination, and each report generated through the report writer required two files—an .FRX and an .FRT. Now throw in a few dozen .DBF, .CDX, and .FPT files, and suddenly even simple applications were back to having hundreds of files in one directory.

Fox Software suggested a solution that's been carried forward by Microsoft: Use separate directories for each type of source code, so that your directory structure for Visual FoxPro looks like **Figure 16.5**.

| Name         | Ext  | Size    | Type        | Date    | Time     | Attr |
|--------------|------|---------|-------------|---------|----------|------|
| ..           |      |         |             |         |          |      |
| Help         |      |         | File Folder | 3/27/99 | 1:02 PM  | d    |
| Include      |      |         | File Folder | 3/27/99 | 1:02 PM  | d    |
| Bitmaps      |      |         | File Folder | 3/27/99 | 1:02 PM  | d    |
| Menus        |      |         | File Folder | 3/27/99 | 1:02 PM  | d    |
| Other        |      |         | File Folder | 3/27/99 | 1:02 PM  | d    |
| Reports      |      |         | File Folder | 3/27/99 | 1:02 PM  | d    |
| Data         |      |         | File Folder | 3/27/99 | 1:02 PM  | d    |
| Forms        |      |         | File Folder | 3/27/99 | 1:02 PM  | d    |
| Libs         |      |         | File Folder | 3/27/99 | 1:02 PM  | d    |
| Progs        |      |         | File Folder | 3/27/99 | 1:02 PM  | d    |
| Tastrade.pjt | .pjt | 30,380  | Microsof... | 7/3/99  | 3:49 PM  | a    |
| Tastrade.pix | .pix | 11,462  | Microsof... | 7/3/99  | 3:49 PM  | a    |
| Tastrade     | .ini | 184     | Configur... | 7/3/99  | 2:01 PM  | a    |
| Mscreate     | .dir | 0       | DIR File    | 3/27/99 | 1:02 PM  | rha  |
| Tastrade     | .app | 825,859 | Microsof... | 5/21/98 | 12:00 AM | a    |

**Figure 16.5.** The standard Visual FoxPro directory structure contains nearly a dozen subdirectories.

Well, I have some problems with this layout. First, I can never keep the names of all of these directories straight. I valiantly struggled with "Why did they spell out REPORTS but abbreviate PROGS?" "Did I call it PROGRAMS or PROGS?" and "Was that in REPS or REPORTS?" And looking through previous versions of FoxPro as well as the default

---

installations of various third-party products, it appears that I wasn't the only person who couldn't be consistent.

Second, I found it ridiculous to have an entire directory for a total of four menu files, or two INCLUDE files, a dozen bitmaps, and so on. For every application you create, it's just that much more work to create and maintain those directories.

Third, and this is more of a real problem than the previous ones, the FORMS/MENUS/PROGS/REPORTS/etc. structure doesn't make any provision for handling libraries and other files used across multiple applications, nor does it provide for a mechanism to deal with data dictionary extensions or other third-party products that require hooks into the app. And it's positively hostile in terms of having to move an app into production and then shuttle changes from the development environment into the production arena.

Finally, you have to either build your application to an .APP or an .EXE in order to run it—which is a real problem if you're making incremental tweaks (remember that attention span issue)—or include all of the subdirectories in the VFP path in order to run from the source code.

Here's what I've come up with as an alternative.

## Developer environment requirements

All but the most trivial of applications will require a number of file types.

The first, pretty obviously, is source code. These are the custom programs, forms, menus, and other files that we have generated and that make up the application. The key distinguishing characteristic is that these files are custom built for this application and they cannot be modified by the user. Note that reports may not fall into this group!

The next group of files make up the common libraries that support all of your applications. In previous versions of FoxPro, these were just Procedure files, but with Visual FoxPro, they include your generic class libraries. Their distinguishing feature is that they are files you created and they're used across applications.

The third group of files include third-party libraries and tools. Smart developers will not try to invent everything themselves, but rather will rely on third-party tools and utilities whenever possible. A number of tools and utilities provide incredible leverage for minimal cost. It's one of the seven wonders of the modern world that a developer would rather spend a couple of months trying to write a barely functional ad-hoc reporting tool when there are several powerful commercial products that provide orders of magnitude more functionality for the cost of a morning of programming time.

Where to keep these third-party tools? I've found it productive to keep them in their own directory instead of throwing them in with the common code. The primary reason is that I tend to update my common code fairly often, but I only add to the third-party libraries every couple months or so.

While I'm on the topic of tools, there's another set of tools that don't become part of an application: "Developer tools" include all those homegrown utilities you've got that make your life easier, as well as commercial products that aid in development. I keep these in the Developer Utilities directory that I mentioned before. I'll cover a few of these in the next section of this chapter.

Next are the data files. These are the tables that hold the user's information. The distinguishing characteristic of these files is that the user has complete control over what goes

into the tables (with, of course, the exception of the key and audit fields). Once a data set has been delivered to the customer, the developer doesn't touch it again unless the structure of the table has changed. (And even then, you should probably provide a utility for automatic update instead of having to depend on human frailty and do it manually.) However, don't think of data as one set of files.

You need to keep multiple sets of data in separate directories so that you can jump from one data set to another. As you get involved in applications that are "big-time," you're not going to want your users practicing on live data. Instead, give them the ability to move between data sets—one set for training and another for actual use. This same mechanism can then be used to run the same application for multiple businesses, and for you, to switch between test and production data sets.

Finally, there are more files that aren't as easily categorized. One type is data files that the user controls but that aren't specific to a single data set. Examples include a table of authorized users or a table that contains user-defined help. Another type is those data files that are created and updated indirectly. For example, many robust applications contain tables that track user logins, application errors, and, in some instances, an audit trail of every change to data. These are also application-specific as opposed to data-set-specific, but the user has no control over them, and, in many cases, isn't even aware that they exist.

A third type of files is the metadata for the application—the data dictionary files. Contrary to popular belief, the database (.DBC) does not make up the entire data dictionary, and you need additional files in your application. Of course, these are application-specific but not data-specific because all of the data sets had better be based on the same metadata!

This has suddenly gotten awfully complex. Perhaps you should just go back to throwing everything back into one directory and then buy a really fast disk controller so the machine doesn't choke on 1400 files in one directory. Well, maybe not...

But how should these files be grouped? To figure this out, I asked two questions: first, to what area of development do the files belong? As you've noticed, three sets of files are application independent, a couple of other sets of files are application-specific but data-independent, and still others are data-specific. Second, who updates the files—the developer or the customer? Because one of the primary problems is updating the production site with modifications, it makes sense to group the files so that they are divided into "updated by developer" and "updated by customer" categories. Here's how we'll do it.

## Drive divisions

Readers of this book will have so much disparity in their machine configurations that it's basically a waste of time to suggest a single methodology of setting up the drive and directory structure and then expect everyone to follow it. However, I'll present one mechanism that can be used on a single machine—even a notebook—but can easily be applied and extended to deal with a network environment. The key point to understand is what I'm trying to accomplish; you can then adapt my techniques or use new ones to apply to your environment.

I have a pet peeve about looking at the root directory of a drive and seeing dozens or hundreds of directories. Not only do the miniature directory list boxes inside most Windows apps make a large number of directories inconvenient to see, but mentally, I can't keep track of more than 15 or 20 entities on the same level. I'd rather break them down into hierarchies.

---

As a result, I segregate software on a machine into at least three separate locations—either drives or directories depending on the hardware of the box:

- The first drive or directory contains the operating system and all shrink-wrap software. As I've mentioned, you should be prepared to reformat that drive often, so keep as little custom stuff on it as you can.
- The second drive or directory contains nothing but custom FoxPro applications and their support files (like third-party tools). If you develop on multiple platforms, you might still want to keep all of your custom development separated from all the other garbage that ends up on your hard disk.

This drive, again, is fairly sparse in terms of the number of directories in the root. The only entries in the root are single directories for each customer, plus directories for application-independent files (common files, developer utilities, etc.) as discussed above. Depending on the size of your shop or of individual applications, you might want to split this across multiple drives on a network.

- Finally, the third drive or directory is used for “stuff that didn’t fit anywhere else.” This includes all internal data, home directories for each user on the network, general reference libraries (like online CD-ROMs), and so on.

## **Root directory contents**

Now it's time to break down the custom apps directory into more detail. As mentioned, there's a directory for each customer, and each project for a customer has its own directory under the customer directory. Before I get into customer directories in any more detail, however, I have to finish up with everything else that's in the root.

The root directory contains three more directories that are application-independent. The first is COMMON60, which contains all common code and libraries that you've written for Visual FoxPro. (Actually, if you went snooping around my network, you'd also find COMMON10, COMMON20, COMMON25, COMMON26, COMMON30 and COMMON50 for the various versions of FoxPro.) The distinguishing characteristic of these files is that you have written them yourself, and they are used across multiple applications.

The second non-customer directory is THIRDPARTY60, which contains all third-party utilities, tools, and libraries that you would use in your application. Many times a third-party utility comes with dozens of files—source code as well as distribution files, sample files, documentation, and other goodies. This directory does not contain all of that—those full-blown installs are contained in their own directories on the Shrinkwrap drive under a directory named something like “FoxPro Utilities.” Rather, this directory contains just the minimum files needed to run the tool.

And just like COMMON, you'll actually find directories named THIRD10, THIRD20, THIRD25, and so on, because typically the libraries don't run across multiple versions of FoxPro.

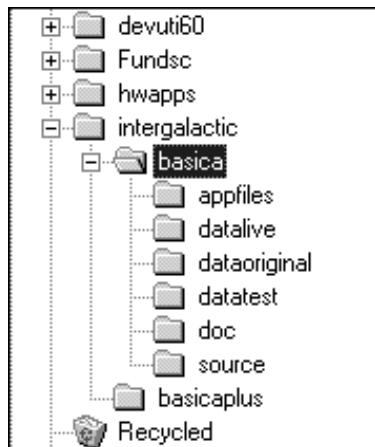
The third root-level directory is called DEVELOPER UTILITIES 60, and again, there are corresponding directories for earlier versions of FoxPro. This directory contains tools and utilities for internal use. Because these files are never taken off-site, I keep the internally

written tools as well as commercial products in the same place and use a naming convention to keep them straight.

## Application directory contents

As described before, each customer has its own directory off of the root directory, and each project for a customer has its own directory underneath the customer's directory.

So what's in the directory for a project? Just a handful, really. See **Figure 16.6**.



**Figure 16.6.** The Intergalactic Corporation has two applications: Books and Software Inventory Control Application, and Books and Software Inventory Control Application–Plus.

At first glance, it might seem that I've just replaced the multiplicity of directories used by Microsoft's Visual FoxPro install with my own—equally voluminous, and clearly more confusing—set. However, a brief explanation will set all right again.

### Doc

I know—it's a radical idea, but I keep all of the documentation for a project. Things like correspondence, specifications, data diagrams, and so on, all end up in this directory. Occasionally you'll have documentation that is specific to a customer, but not to any specific project—in those cases, I create a generic DOC directory on the same level as each project.

### Source

SOURCE, obviously, contains source code. All of it. Into this directory goes the project, every custom menu, program, form, and class library file (classes that are specific to this project), as well as any other types of files (like bitmaps) for this application. I've already whined about my dislike for multiple source code directories, and you're welcome to create separate directories if you like; you can still use the rest of this structure with very little modification.

---

One question that has already come up has dealt with the number of files that end up in SOURCE in a big application, and the resultant drag on performance. Actually, however, there really isn't a drag. I've built applications with more than 1000 source code files on a 266 MHz PII and really haven't noticed much of a drag. Applications delivered to customers are compiled to a small set of .EXEs and .APPs, and they're placed above the SOURCE directory, so the effects at the customer's site (where they probably have lower-powered machines) aren't noticeable either.

## **Appfiles**

APPFILES contains all application-specific files for the system. This consists of the data dictionary and any other files that are application-specific, whether or not the user will be able to modify them—either intentionally (such as a user-defined help file or the users table) or unintentionally (those files that the system will update behind the scenes, such as the logins table and the error-tracking table). The key attribute of each of these files is that they are data-independent—the same information applies regardless of which data set the user is working with. I used to keep metadata files in a separate directory, but Visual FoxPro's architecture with the .DBC has made that unnecessary.

Why? Note that the .DBC file for a dataset does not belong in the APPFILES directory. Due to the nature of how Visual FoxPro handles the database, the best design decision is to have a separate .DBC for each set of data. It's not that bad; because the .DBC stores the relative path of its tables (if they're all on the same drive), you could create a .DBC, add the tables, and then just copy the whole structure to another directory on the same level. When your application runs, it will switch to the appropriate data directory and open the database.

The only glitch is making sure that forms that have been tied to a specific data set will recognize the current data set, but I've already discussed the code in the BeforeOpenTables event of a form's Data Environment required to make this happen (Chapter 15, if you skipped over it.) It's probably good to mention a second time, though, that when you build your forms, the database you use initially will be bound to the form. For this reason, I always bind all forms to the database in the DataOriginal directory. (More on this when I cover the data directories.)

What types of files might you have in APPFILES?

- A\_USER contains one record for each user of the system. The record contains the user's full name, login name, password (encrypted, of course), permission level, a flag for whether or not they're allowed to currently access the system, and fields for storing individual user preferences.
- A\_PROC contains one record for each time a user logs in or out of the system. The record contains the user's login name, the date and time logged in, and the date and time logged out. This file can be used for three purposes. First, you can monitor access to the system—seeing who is using the system and when. This can be useful information both for security problems as well as to track down bugs that might be user-related. Second, if you suspect users are not exiting the application properly, simply turning their computers off ("Oh, no, I never turn my computer off without logging out first!"), you can track them down by looking for entries that have a login but no matching logout. And finally, you can do a quick query against this file to see

who might be logged in. Note that this is not foolproof, because a user who just shut their computer off will still appear to be logged in.

Note that a single record is used to record the matching login and logout. It's simple to write a routine that will locate the last record for the current user that doesn't have a logged-out date and time, and update those fields during their logout procedure.

- A\_EVENT contains one record for each instance that the application's error handlers have been fired. The record contains the number and the description of the error, the line number (and line, if available) of source code that was executing at the time, the login name of the user, and the date and time the error occurred. The associated memo file contains a text file that lists the status of the application and environment at the time of the error.
- A\_HELP is the compiled HTML Help file for the system. See Chapter 25 for information on building HTML help files.

### **Data directories**

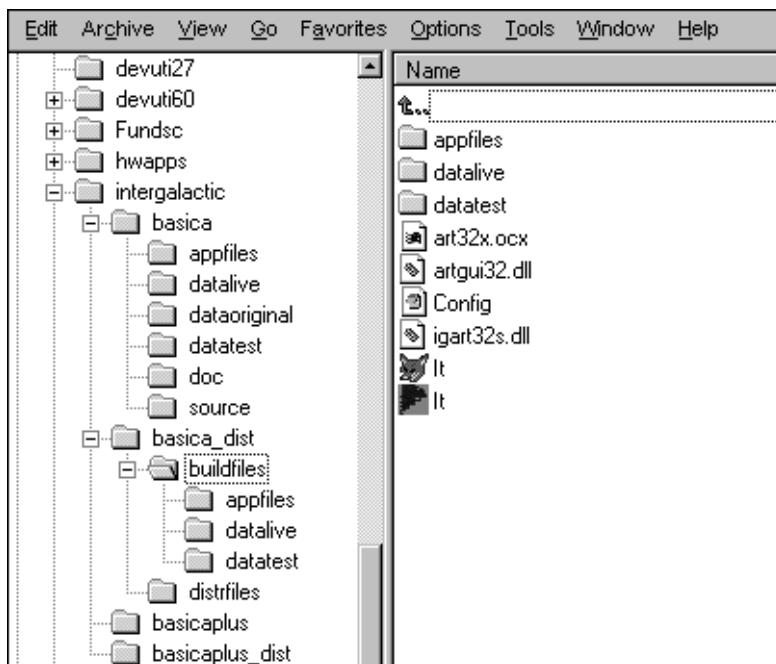
Finally, I keep at least two (and usually three) data directories with a project. I reverse the names ("DataOriginal" instead of "OriginalData") so that all of the data directories show up in proximity in Explorer. DataOriginal contains the dataset I use to build the application's forms and classes. In other words, when I create a form, I use DataOriginal as the source for the tables. DataOriginal actually contains no data, which serves an important purpose. I get to make sure that the application will run even without records in the tables—and that is often a handy test. It's so embarrassing to build and test an application, and then, as the last step, zap all of the tables and ship the system—only to have customers call with bug reports because the app can't handle empty tables.

Each data set contains the .DBC as well as a few specific files—IT.DBF contains the last used key value for each table in the data set as well as other values that are automatically incremented by the system, and ITLOOK is the standard lookup table. This table is not considered to be data-independent because its values might be specific to a particular data set: A user in the training data set might create a bunch of garbage values that shouldn't be accessible by the production data; or multiple businesses might have different values in their lookup tables.

### **Build directory contents**

Developing apps is one thing—with Visual Studio 6, distribution is a lot more complicated. It's not enough to simply create an .EXE and ship it to your customer with a couple of runtime files. When you ship a VFP application, you'll need to build a complete set of distribution disks, complete with a standard Windows setup program.

I mentioned earlier that each customer directory contains separate directories for each project. However, because of this involved distribution process, you'll need a second set of project directories to hold your distribution files, as shown in **Figure 16.7**.



**Figure 16.7.** The distribution directory for the Books and Software Inventory Control Application project has subdirectories for storing files to build from and for storing the resulting distribution files.

The first subdirectory under the project's distribution directory is BUILDFILES. It is used to store a copy of the application from which the distribution disks will be built. This directory will hold the application as the end user will see it; thus, you'll have an APPFILES directory, and a test and a live data directory (but not DataOriginal). You'll also want the .EXE you built for your application, an icon file, and, optionally, a config file. You don't need to include the VFP runtime—the Visual FoxPro setup wizard will automatically grab it from the Windows system directory.

Finally, you'll need a target for the setup wizard—where it will put the distribution files for your application. I use the DISTRFILES directory under the project's distribution directory as a handy place to keep them organized. If you like, you could even create multiple subdirectories under the DISTRFILES directory for each build you produce.

## Developer utilities

I'm of two minds while writing this section. One part of me wants to share a bunch of cool utilities that I think you'll find useful. The other part of me knows you all too well—and I know you're thinking, "Building my own utilities is half the fun of being a developer!" So I'm going to take the middle road and list a few simple tools I've found handy, and then describe a few more that you might want to build yourself.

I've divided this section into two pieces: those that I wrote myself, and those that I found elsewhere.

## My own stuff



Part of the thrill of being a developer is writing your own utilities in response to needs that come up during your day-to-day coding. Here are a couple of mine and Doug's. They're included with the source code files for this book at [www.hentzenwerke.com](http://www.hentzenwerke.com).

www

I'm forever using a Wait window or a message box to quickly display the value of some memory variable or contents of a field. The problem is that you need to convert the value to a string in order for it to display properly—and, if that's not hard enough to type, you don't always know what the data type of the value is. (By doing this in the first place, you already demonstrate that you don't know what the value is—it's likely that you don't even know what the data type of the value is, right?)

This little function—only two characters long—takes any type of data and displays both the type of data and the value itself in a Wait Window box in the center of the screen. You can pass a “,1” parameter to WW to display the value in the upper right corner of the screen, where Wait Window displays by default, or a “,2” parameter to display the value in the upper left corner:

The parameters correspond to the quadrant numbers you learned about in high school geometry. See **Figure 16.8**.



**Figure 16.8.** “WW” will produce a Wait window that displays the data type and value of any variable or field name passed to it.

## Array Browser

You can use the Debug window to examine values of an array. But if there are more than a few values in a two-dimensional array, the display is hard to use because you just get a single long list of every array element. I wrote the Array Browser many moons ago in order to display an array in a visual row-and-column format.

You can call it three ways. If you simply call the routine, you'll be prompted for an array name:

```
=ab6()
```

If you pass the name of an array, a form will be opened with the array displayed in one of them-thar ole-fashioned browse winders:

```
=ab6("aMyArray")
```

## DevHelp

Just about every developer I know has about 1500 Post-It Notes stuck around the edges of their monitor, reminding them how to do this and that. The problem with this system is that it's not multi-user. So I created a little table on one of the network drives that everyone could access—and in it, I started loading all sorts of tips and how-to's that had been stored on layers of yellow stickies. Then I wrote a little routine that opened this table from the Dev menu pad (which I discussed in the "Startup program" section earlier in this chapter).

## OpenAll

Are you as lazy as me? For some odd reason, I find myself constantly opening a bunch of tables in a database. And it's a real pain:

```
Use DBF1 in 0
Use DBF2 in 0
Use DBF3 in 0
Use DBF4 in 0
Use DBF5 in 0
Use DBF6 in 0
...
... (all the way till)
use DBF322 in 0
```

So I wrote this little program that digs out the names of all the .DBF files in the current directory, and opens each one of them.

## Z

You know when you're trying to track down a really nasty bug? And you've tried everything? What's the very last thing you do before quitting FoxPro, turning your machine off, and then starting back up again? You go through a series of commands that should shut down everything in sight:

```
close all
clear all
clear
on error
on escape
```

and so on. Well, in order to save a few keystrokes (and the last few strands of hair on my head), I wrote a program that does all this—and much more—to clean up the environment and set everything back to where VFP had it when it was last loaded. And because it's now the last thing I do, I named the program "Z". Then, after any kind of error during my programming or testing, I simply:

```
do z
```

and everything is set back to the way it should be.

### **HackCX**

Once you get comfortable with the idea that forms and class libraries are simply tables—and, even better, that there's some rhyme and reason to how they're organized—it's just a short warp jump to realizing that you can open and modify them yourself.

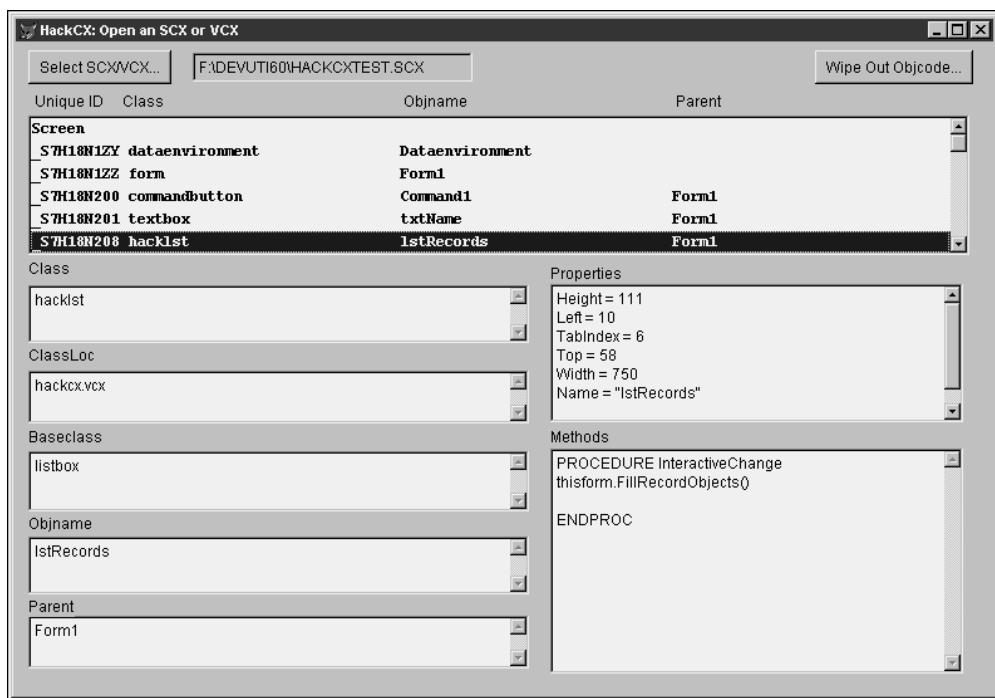
For example, suppose you've got this complex data-entry form with a couple of page frames and dozens of other controls on it. And suddenly it hits you, when you're wondering why the three text boxes on the third tab aren't behaving like you were expecting, that those came from VFP's base classes, not from your own class library.

After you've recovered from the slap upside the head you gave yourself, you wonder how you're going to fix the error without going through every control, one by one, to determine whether it's from the proper class. Then it dawns on you—you can just open the .SCX and change the Class field from "textbox" to "hwtxt" and Class Location field from "" to "hwctrl62.vcx".

And *then*, after you've done this a few times, you realize that it's really clunky—having to open the form or class library as a table, then browse and resize the window, and then open a half-dozen or more memo fields to get at the data you want. Wouldn't it be better to have a nice user interface to do this?

HackCX is a simple form (and associated class library) that does just that: opens an .SCX or .VCX, allows you to scroll through every record, and allows you to see the contents of and make changes to an object's class, class location, base class, object name, and parent. See

**Figure 16.9.** You can even see and modify the properties and methods. However, because the compiled code for those items is stored in the Objcode field, you'll want to make sure to use the Wipe Out Objcode button—in effect deleting the compiled code from the form or class library, and thus forcing VFP to recompile the next time the form or class is used.



**Figure 16.9.** HackCX allows you to view and edit multiple memo fields in an .SCX or .VCX.

**NIHBSM (Not Invented Here But Still Marvelous)**

## **.CHM files**

The first thing that should be in your Developer Utilities directory is the complete set of .CHM files from the books that I publish. (What? You mean you don't have all of them already? I'll wait right here while you go place your order for the rest ... really, it's no trouble ... go ahead, do it now, before you forget *again*!) My Dev menu pad has shortcuts to all of the .CHM files, as shown in **Figure 16.10**.

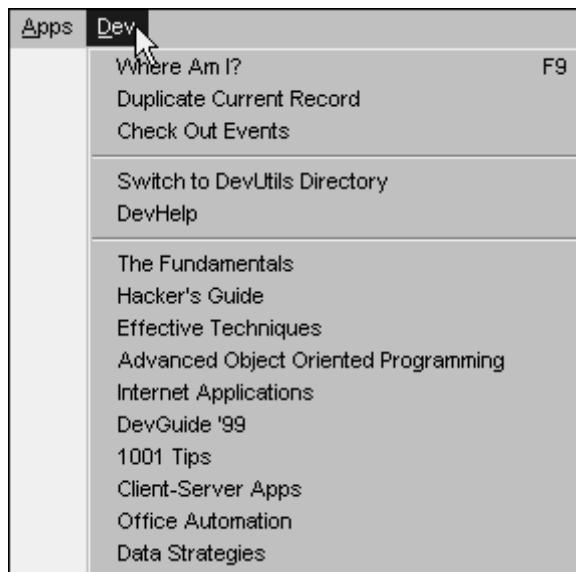
In the CASE statement in GOFOXGO, then, I make a parameterized call to a generic help function:

```
=1 help("HACKFOX")
```

The generic help function looks like this:

```
*****
func l_Help
*****
lparameters m.tcNameCHM
m.lcCurrentHelp = set("Help",1)
set help to (m.tcNameCHM)
help
if !empty(m.lcCurrentHelp)
  set help to (m.lcCurrentHelp)
endif
return .t.
```

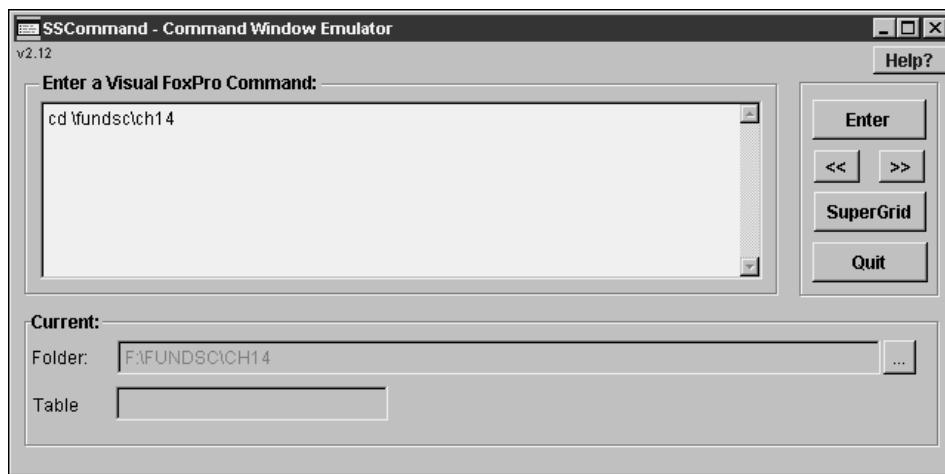
This way, you don't blow away the existing Visual Studio help reference but you still have one-click access to any of the .CHM files.



**Figure 16.10.** My Dev menu pad includes commands to launch all of the Essentials .CHM files.

### SoftServ's Command Window Emulator

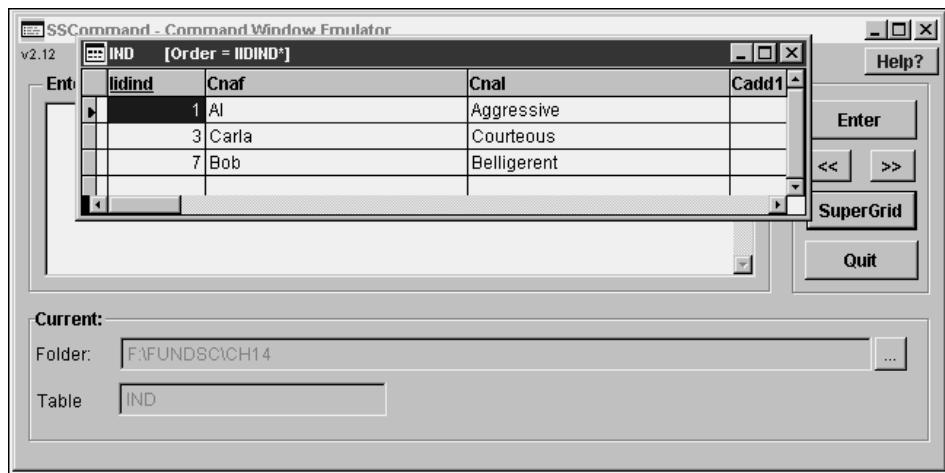
SSCommand was designed to help in situations where you have a VFP application running (compiled) at a client site, and the client does not own Visual FoxPro. It emulates the VFP Command window (see **Figure 16.11**), allowing you to perform virtually any FoxPro command. It frees you as a developer from having to tote your notebook computer when you have to make on-site data maintenance or report modifications. It also makes life much easier if you use remote dial-in software for application maintenance.



**Figure 16.11.** SSCommand from SoftServ.

Like the Command window, the Command Window Emulator maintains a list of your previous commands. It also allows for multi-line commands using the semicolon for line continuance. You can browse a table, modify a table's structure, modify a report, run a report, and so on. The only thing you *can't* do is compile source code.

In addition to providing all the regular functionality of the Command window, they've created one dandy data-manipulation tool. It's called SuperGrid. See **Figure 16.12**. This is a browse-like grid that has many features, including "on-the-fly" column indexing, actual field name column headers (remember the old days?), column partial-value searching, column locate-value searching, compound index creation, and so on.



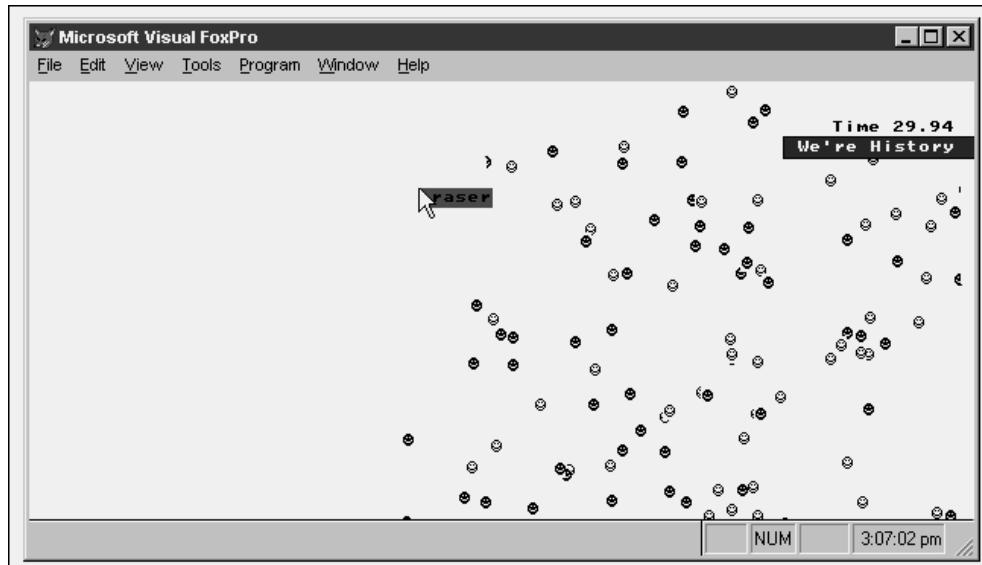
**Figure 16.12.** SuperGrid from SoftServ.

### Eraser

Okay, so this is silly, moronic, and makes no sense. But so is going to work on Fridays. I found it on one of the CompuServe forums a hundred years ago, and have always kept it around for idiot relief. See **Figure 16.13**.

When you run Eraser, it fills your VFP screen with happy faces, and gives you a timer and an “eraser” (a cursor about four characters wide). As soon as you start moving the eraser (to erase the happy faces, don’tcha know...), the timer starts running. The goal, of course, is to erase all of the happy faces in the shortest amount of time possible.

You’ll want to close all of the windows in VFP first, and then run Eraser from the Program, Do menu command.



**Figure 16.13.** The Eraser game is just plain silly.

### Third-party applications

By now, hopefully you’re all fired up and have a ton of ideas for your first Visual FoxPro application. It’s tempting to find a quiet room, fill the refrigerator with Jolt cola, and lock yourself in until the app is finished. But you shouldn’t do it alone. In fact, the ugly truth is that you can’t do it alone anymore. Visual FoxPro is too big and too complex for most people to master in a short amount of time. And while you could spend months and years learning the product, chances are you need to get applications up and running now! (If not sooner.)

Well, there’s no need to reinvent the wheel or to do everything yourself. There are a number of third-party products that provide various mechanisms and tools to help you develop applications, and I’m going to steer you toward 17 of the very best. In each case, the third party has invested thousands of hours of development into creating robust, feature-rich products that

are used throughout the world. For a couple hundred bucks, there's no way you could match the functionality and reliability that these products offer. I've known the principals of each of these firms for years—and they're all individuals of the highest caliber. They stand behind their work, and the rest of the industry would do well to emulate their business practices. And no, I'm not getting commissions or anything. (In fact, I think I owe a couple of them a beverage or two the next time I see them...) It's just that I think these are all great products and they deserve your consideration.

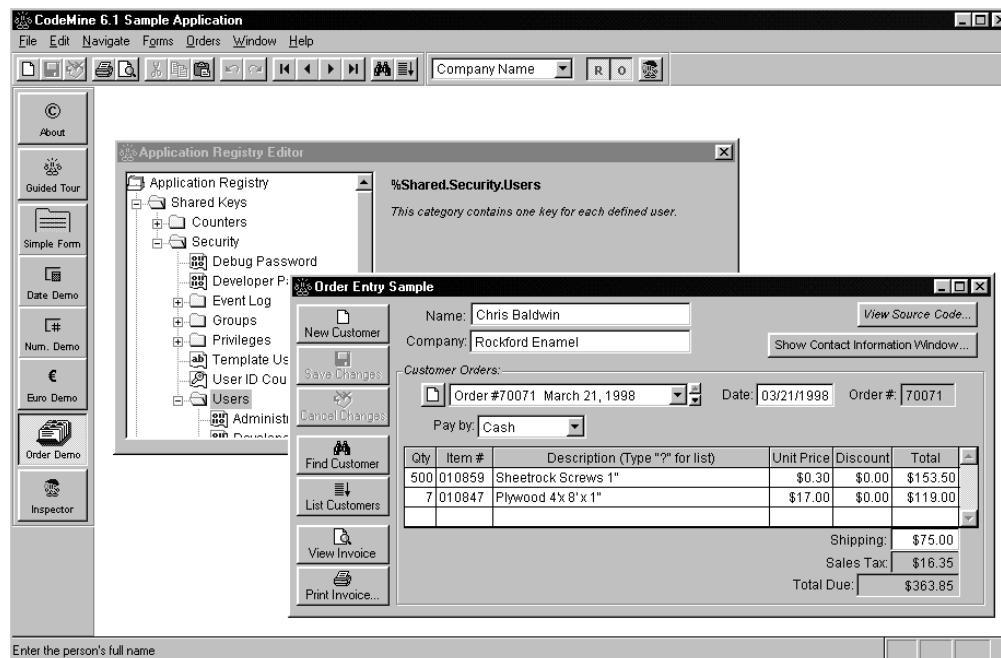
And by the way, just because I mention only these doesn't mean that everything else out there is junk. These are important tools that every new Visual FoxPro developer should be aware of. There are dozens more tailored for specific needs or certain functionality, but they aren't as universally applicable.

I've included a brief description of each (in most cases, lifted directly from the company's marketing literature) to get you started. Contact them at the sites below and they'll be happy to send you piles of literature and demo disks, and also to answer any questions you have.

## CodeMine

CodeMine is a high-performance, object-oriented application framework for Visual FoxPro 5.0 and 6.0. CodeMine includes a comprehensive set of integrated class libraries and development environment extensions for the ultimate in Rapid Application Development. See **Figure 16.14**.

[www.codemine.com](http://www.codemine.com)

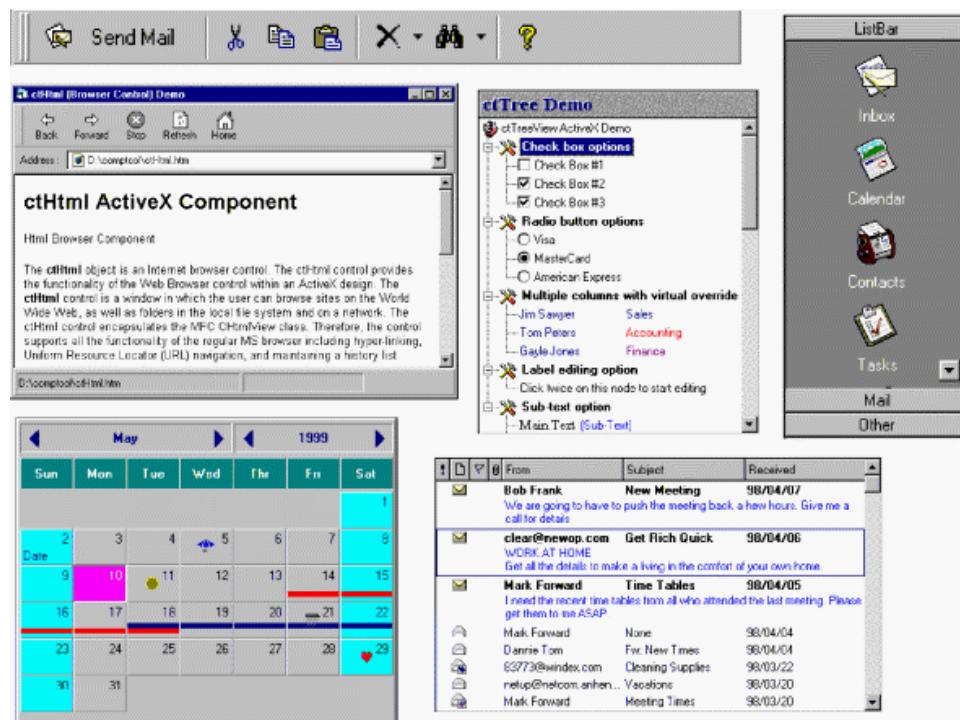


**Figure 16.14.** The sample application for CodeMine.

## DBI Technologies ActiveX controls

DBI has been at the forefront of producing ActiveX controls that work in Visual FoxPro. Many ActiveX control vendors concentrate their efforts on the Visual Basic market, and thus many controls have difficulty running properly in VFP. DBI is one of the few vendors that has focused on the VFP market, making sure their controls behave properly in both VB and VFP. See **Figure 16.15**.

[www.dbi-tech.com](http://www.dbi-tech.com)

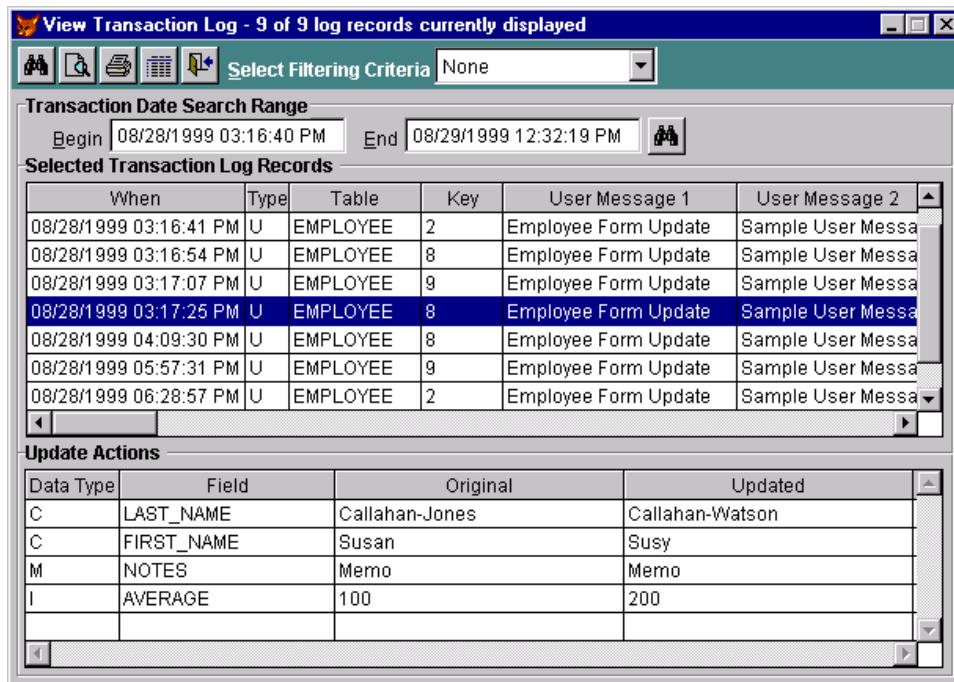


**Figure 16.15.** The Component Toolbar from DBI.

## FoxAudit

FoxAudit, from Take Note Consulting, allows developers to add complete, automatic, client-server-like, transaction logging and audit trail support to their applications. It is implemented as a Visual FoxPro class that captures information each time a user inserts, updates, or deletes records in an application. The details of each update are stored in a transaction log table. See **Figure 16.16**.

[www.takenote.com](http://www.takenote.com)



**Figure 16.16.** Viewing a transaction log using FoxAudit.

## Foxfire!

It's the rare application that doesn't require output, and it's even rarer when you find a user who doesn't want to change "one little thing" on the reports you've put together. Many developers decide they want to create a generic ad-hoc report writer that will allow users to create and store their own reports. A few months later, the developer emerges from his office, dark circles under his eyes, muttering, "If only I could get this one thing to work, I think I'd be done!" Foxfire!, from Micromega Systems, provides this functionality and more. See **Figure 16.17**.

[www.micromegasystems.com](http://www.micromegasystems.com)

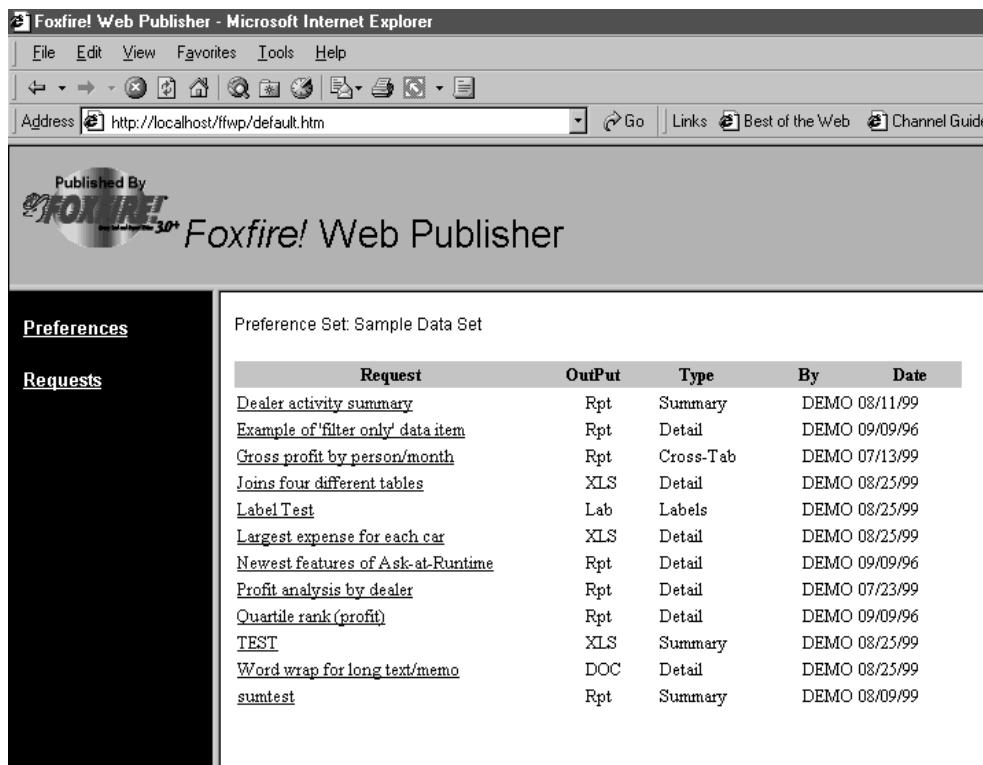


Figure 16.17. Foxfire! from Micromega Systems.

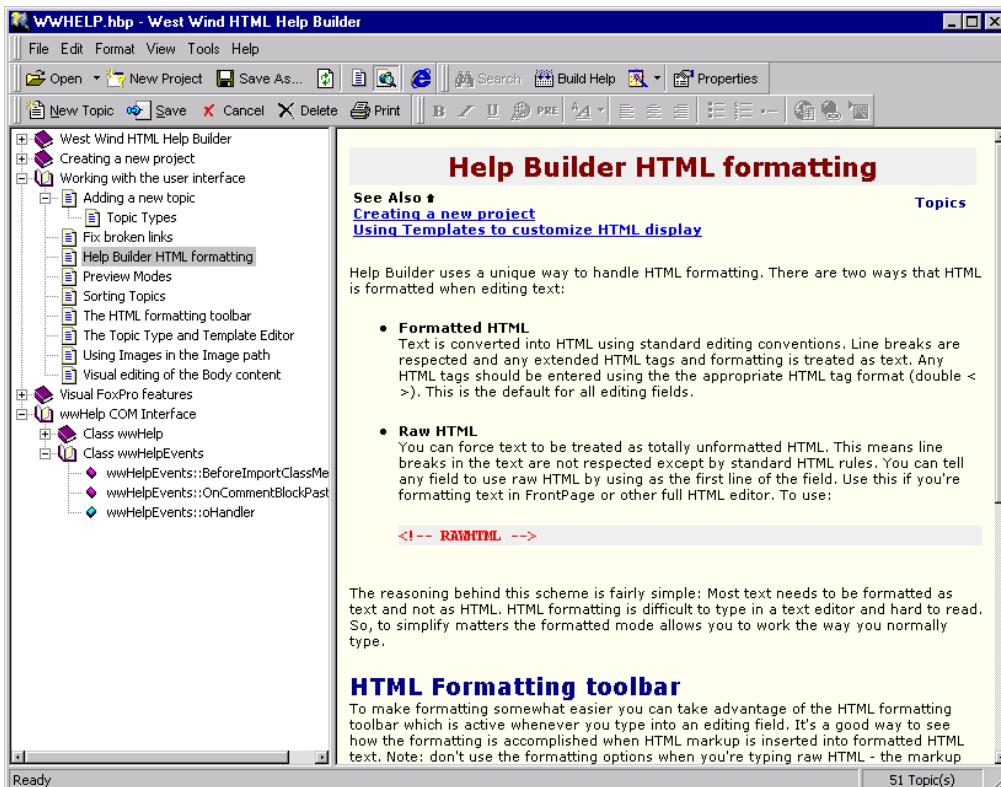
## HTML Help Builder

West Wind HTML Help Builder offers a new way to build developer help files. Most help file-generation software focuses heavily on the visual aspects of help generation, so you end up using a word processor like Word with macros to generate your content. That works well for end-user documentation—but what about developer documentation? Developer documentation tends to be much more structured and uses repetitive data entry that's often easier to accomplish via structured data input. With this structured approach it's easy to integrate the help file creation into the development process, both for creating end-user developer documentation as well as building internal documentation and references. See **Figure 16.18**.

Help Builder also focuses on laying out the structure of the help file. All while you're working with Help Builder you're creating a layout that matches the layout of the HTML Help file and you can see the project take shape as you work. All views are live so you can see any text you create immediately as HTML, and the project structure is at any point visible in real time. All topics inside the viewer are also live, so you can click and move around the help file just like you can in the compiled HTML Help file. Finally, when you're ready to build your output, Help Builder allows you to create both stand-alone HTML documents or an HTML

Help output file. This way you can use the output both for inclusion with your application as well as display the output on the Web without any special viewers.

### [www.west-wind.com](http://www.west-wind.com)



**Figure 16.18.** *HTML Help Builder from West Wind Technologies.*

## INTL

Steven Black's INTL Toolkit for Visual FoxPro 6 works either at run-time or at compile-time. Run-time localization gives you one executable for the world, with resources bound at run-time. INTL for VFP 6 has the following run-time capabilities:

- Interface strings
- Fonts
- Images
- Data sources

- Currencies
- Right-To-Left writing systems
- Dynamic dialogs

Compile-time localization, which is new to INTL for VFP, gives you one executable for each locale, with resources bound when you build your application. INTL for VFP 6 has the following compile-time capabilities:

- Interface strings
- Fonts
- Images
- Data sources
- Dynamic dialogs

If you do multi-lingual work, you must have this product!

[www.stevenblack.com](http://www.stevenblack.com)

## **Mail Manager**

Mail Manager, by Classy Components, is a MAPI-compliant component that allows you to easily integrate e-mail into your applications. When coupled with QBF Builder (see below) or with a compliant result set, it will allow you to send an e-mail to a queried result set of recipients—for example, “Send a welcome e-mail to all new customers.” See **Figure 16.19**.

[www.classycomponents.com](http://www.classycomponents.com)

## **Mere Mortals Framework**

The Codebook methodology has been popular among developers for several years. Kevin McNeish has taken the complexity out of it and created an application framework that, as its name suggests, “mere mortals” can use. The Mere Mortals Framework, from Oak Leaf Enterprises, is a robust tool for rapidly developing flexible and adaptable applications. It helps you understand and use new technologies while providing a clear migration path that protects your investment in existing applications. See **Figure 16.20**.

With Mere Mortals you can create applications that anticipate changes in the user’s requirements and the software industry. The Framework, along with extensive documentation, guides and educates you throughout the development process of creating true three-tier, client-server applications that are easily adapted to run over the Internet. Mere Mortals also takes advantage of the Visual FoxPro Component Gallery by integrating and enhancing the Foundation Classes.

[www.oakleafsd.com](http://www.oakleafsd.com)

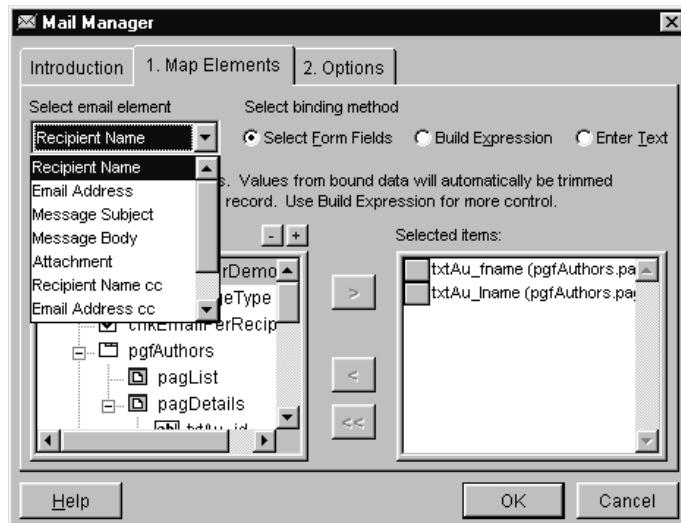


Figure 16.19. The Mail Manager interface from Classy Components.

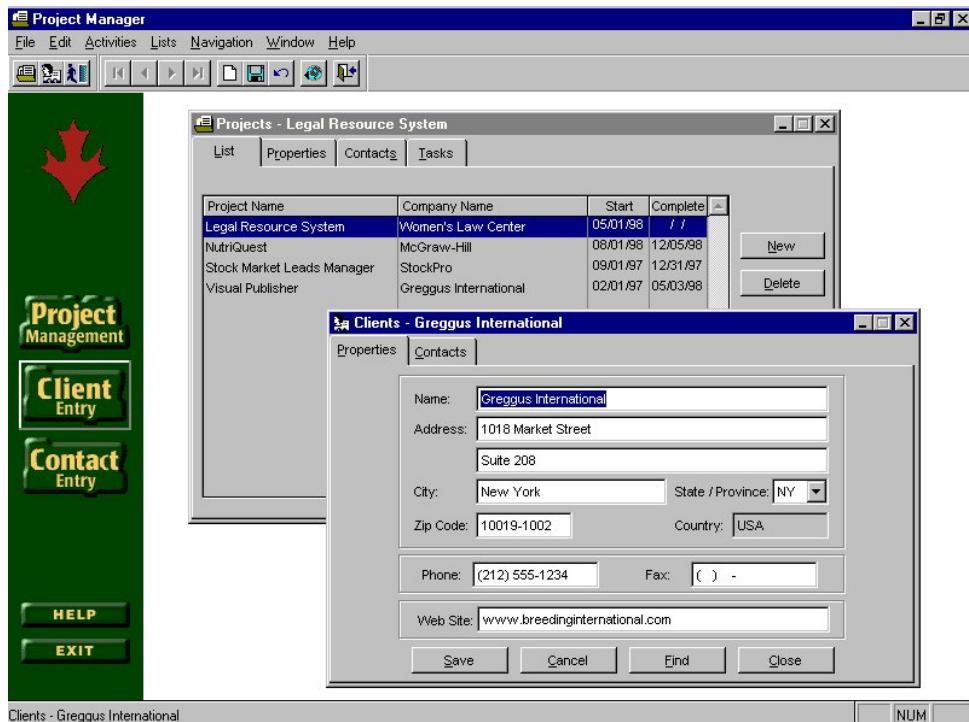
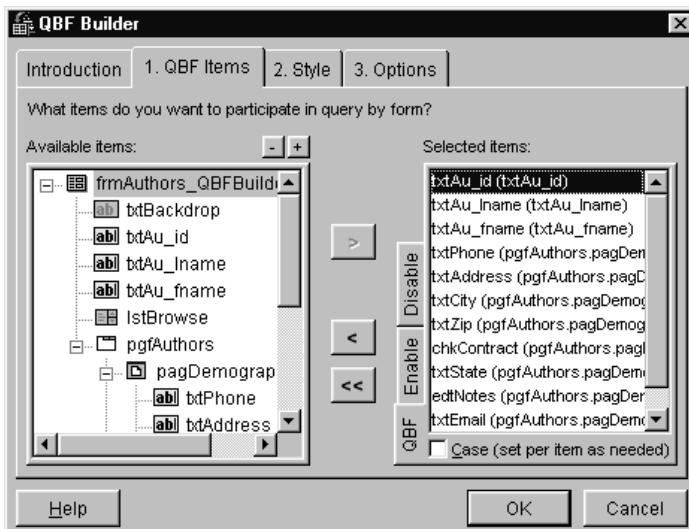


Figure 16.20. Mere Mortals Framework from Oak Leaf.

## **QBF Builder**

QBF Builder, by the same folks who brought you Mail Manager, is a control that you can drop on any data-entry form to provide “query by form” capabilities. It works with tables and local/remote views. You can save and rerun query definitions, and access advanced criteria (“contains”, “inlist”, “like”, etc.) via a context menu on the query form. See **Figure 16.21**.

[www.classycomponents.com](http://www.classycomponents.com)



**Figure 16.21.** QBF Builder from Classy Components.

## **Stonefield Database Toolkit**

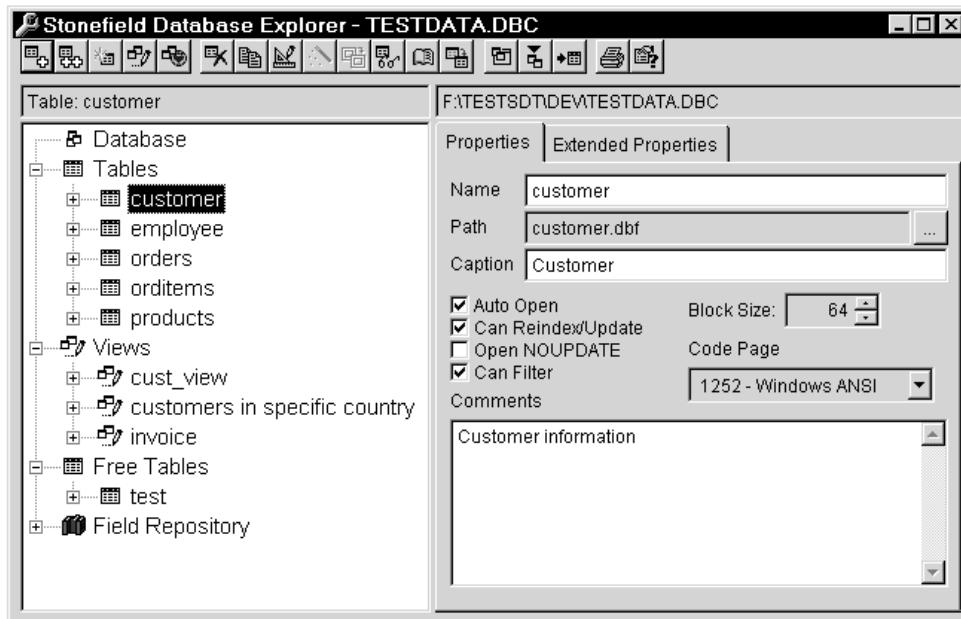
Visual FoxPro provides something FoxPro developers have needed for years—a built-in data dictionary. Visual FoxPro’s data dictionary provides table and field validation, field captions, triggers, even table and field comments. See **Figure 16.22**.

Unfortunately, many things are missing from the data dictionary, such as structural information necessary to create or update table structures at client sites, and useful information such as captions for tables and indexes. Also, the tools Visual FoxPro provides to manage the database container itself are less robust than you’d expect. For example, altering the structure of a table breaks views based on that table. Table and field-name changes aren’t propagated throughout the database, resulting in orphaned database objects. Moving a table to a different directory or changing the .DBF name causes the database to lose track of the table.

Stonefield Database Toolkit (SDT) overcomes these limitations and many others we’ve found in Visual FoxPro, and provides additional functionality that serious application developers need. There are three aspects to SDT:

- It enhances the tools Visual FoxPro provides to manage the database container.
- It provides the ability to define extended properties for database objects and set or obtain the values of these properties at run-time.
- It includes a class library you can add to your applications to provide database management functions at run-time, including recreating indexes, repairing corrupted table headers, and updating table structures at client sites.

[www.stonefield.com](http://www.stonefield.com)



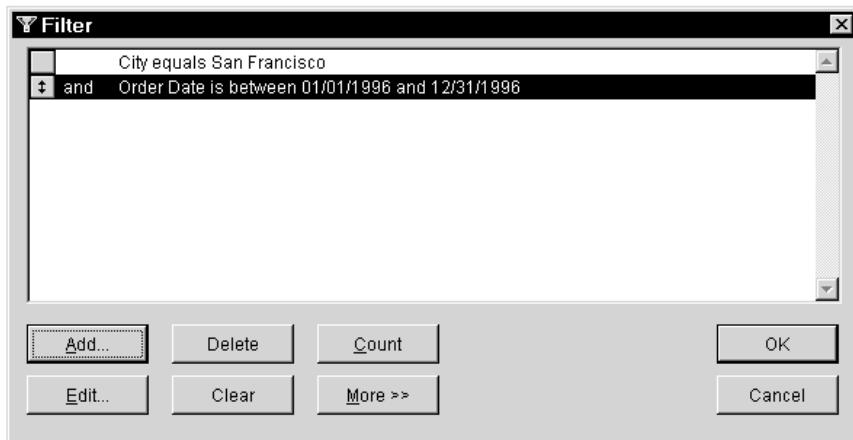
**Figure 16.22.** Stonefield Database Toolkit from Stonefield Systems.

## Stonefield Query

Stonefield Query is a powerful end-user query-builder tool. Users can specify fields and operators from multiple tables using English descriptions rather than cryptic names and symbols, without knowing about complex stuff like join conditions. See **Figure 16.23**.

It's easy to implement as a developer: simply drop an SFQuery object on a form, set some properties, and call the Show() method to display the Filter dialog. You can use the cFilter property to set a filter or as the WHERE clause in a SQL SELECT statement, or use the DoQuery() method to perform the SQL SELECT.

[www.stonefield.com](http://www.stonefield.com)



**Figure 16.23.** Stonefield Query from Stonefield Systems.

## Stonefield Reports

Stonefield Reports is an end-user report manager/writer. It sports a simple “wizard” interface—after selecting a report from the TreeView list of reports, your users can select the sort order, enter filter conditions, and select where the output should go (printer, disk file, spreadsheet, screen preview, and so on). See **Figure 16.24**.

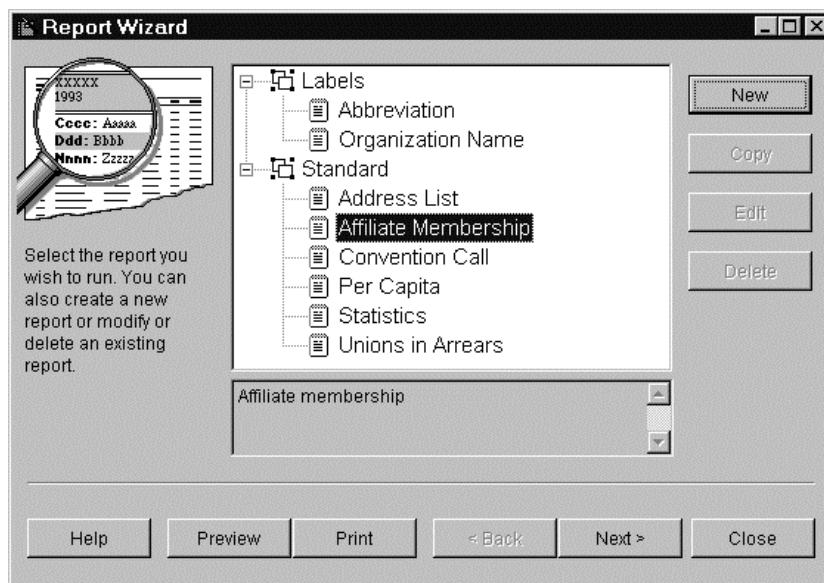
In addition to running predefined reports, you can teach your users how to create their own “quick” reports in just minutes. They simply select which fields to report on from the list of available fields (full-text descriptions, of course), and they’re done! For finer control, they can select from a “Field Properties” dialog how each field should appear, including column heading, grouping, and totaling.

[www.stonefield.com](http://www.stonefield.com)

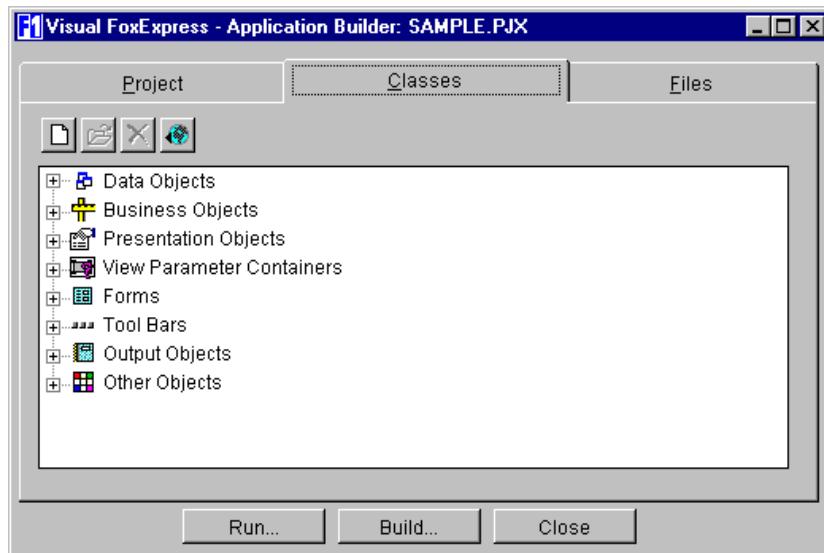
## Visual FoxExpress

The Visual FoxExpress Framework, from F1 Technologies, is the first complete n-tier solution for Visual FoxPro that’s suitable for all types of application development. It’s a framework that’s lean enough to deliver the performance your applications require and flexible enough to allow you to easily incorporate the complex business rules they need. It separates user interface, business rules, and data. It’s possible to create user interfaces that are not directly bound to data at all. VFE apps can be deployed as COM or DCOM objects with no user interface, accessible from other front-ends such as ASP or DHTML Web pages, or built with other tools such as Visual Basic. FoxExpress generates 100% Visual FoxPro code, and if you need to get applications up and running quickly, you should have this in your bag of tricks. See **Figure 16.25**.

[www.fltech.com](http://www.fltech.com)



**Figure 16.24.** Stonefield Reports from Stonefield Systems.

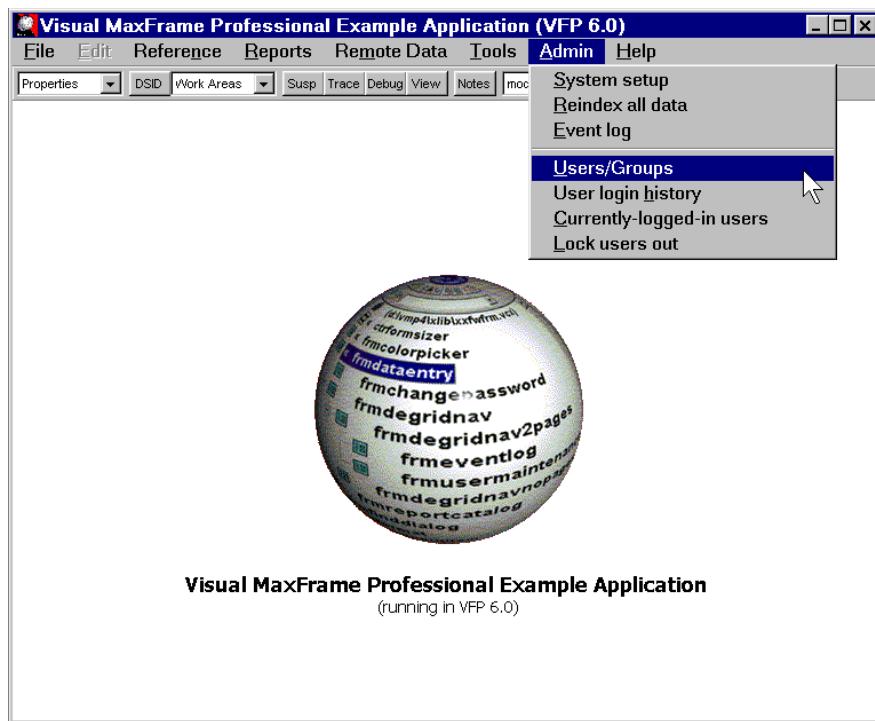


**Figure 16.25.** Visual FoxExpress from F1 Technologies.

## Visual MaxFrame Professional

Designed and produced by VFP expert Drew Speedie, VMP is an application framework for VFP and is now in its fourth version. It includes complete source code, extensive documentation, and a sample application. Version 4.0 includes client-server services, an optional security system, new and enhanced developer tools, as well as a wealth of improvements through the product. See, **Figure 16.26**

[www.maxlink.com/vmpmain.htm](http://www.maxlink.com/vmpmain.htm)



**Figure 16.26.** Visual MaxFrame Professional from Metamor Worldwide.

## Visual Web Builder

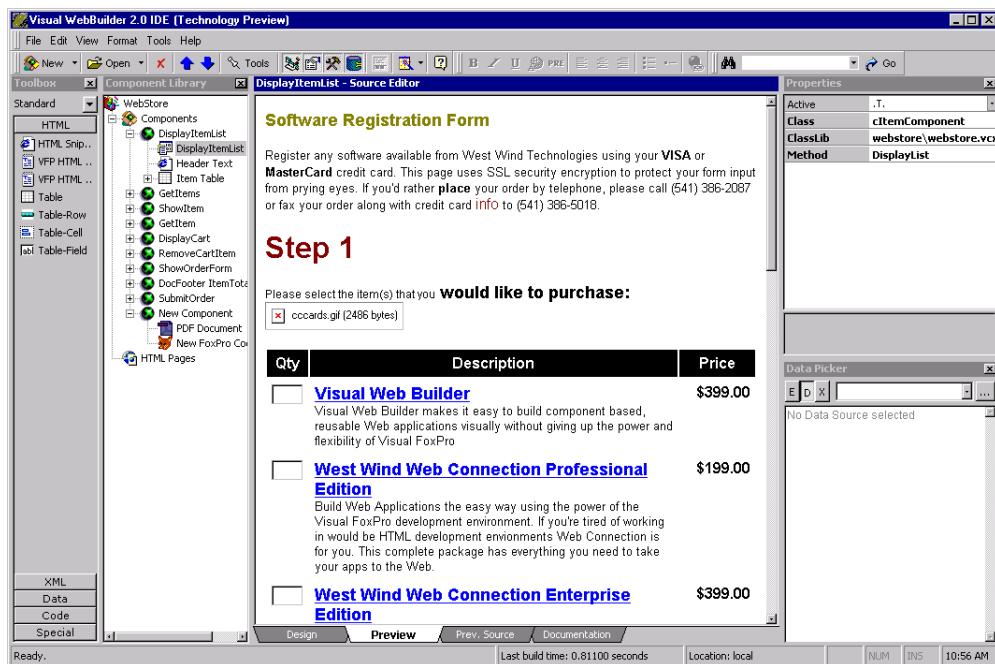
Are you ready to build Web- and HTML-based application the easy way? Need to plug HTML functionality into existing desktop applications for a rich user interface? Need to build back-end Web applications? Then check out Visual WebBuilder from West Wind Technologies and get ready to jump into the world of visual Web development using HTML, scripting, and Visual FoxPro code in a highly visual and interactive environment. See **Figure 16.27**.

Visual WebBuilder leverages the power of Visual FoxPro both internally in the framework as well as extending this power to your applications—you can run full FoxPro code inside

FoxPro classes, program files, and scripted HTML pages that mix HTML and FoxPro code and expressions.

A Visual Component Editor allows you to build small components that can interact with each other. Take code reuse to a new level by creating small, functional components that can be plugged into other components or be directly accessed from Web pages. Whether you use a Visual FoxPro back-end tool like West Wind Web Connection or Active Server Pages, Visual WebBuilder has you covered.

[www.west-wind.com](http://www.west-wind.com)



**Figure 16.27.** Visual WebBuilder from West Wind Technologies.

## West Wind Web Connection

Rick Strahl's West Wind Web Connection is the premier tool for building Web applications with Visual FoxPro. See **Figure 16.28**.

Web Connection provides a scalable Web platform for building fast, powerful and flexible applications with ease. Take your pick between code-based HTML creation or FoxPro-based HTML scripting, dynamic form rendering or PDF Report generation to create your Web output. The supplied classes and tools greatly simplify receiving Web server requests, using the provided information and generating the final dynamic HTML and other data content like XML or raw data to return to the client.

[www.west-wind.com](http://www.west-wind.com)

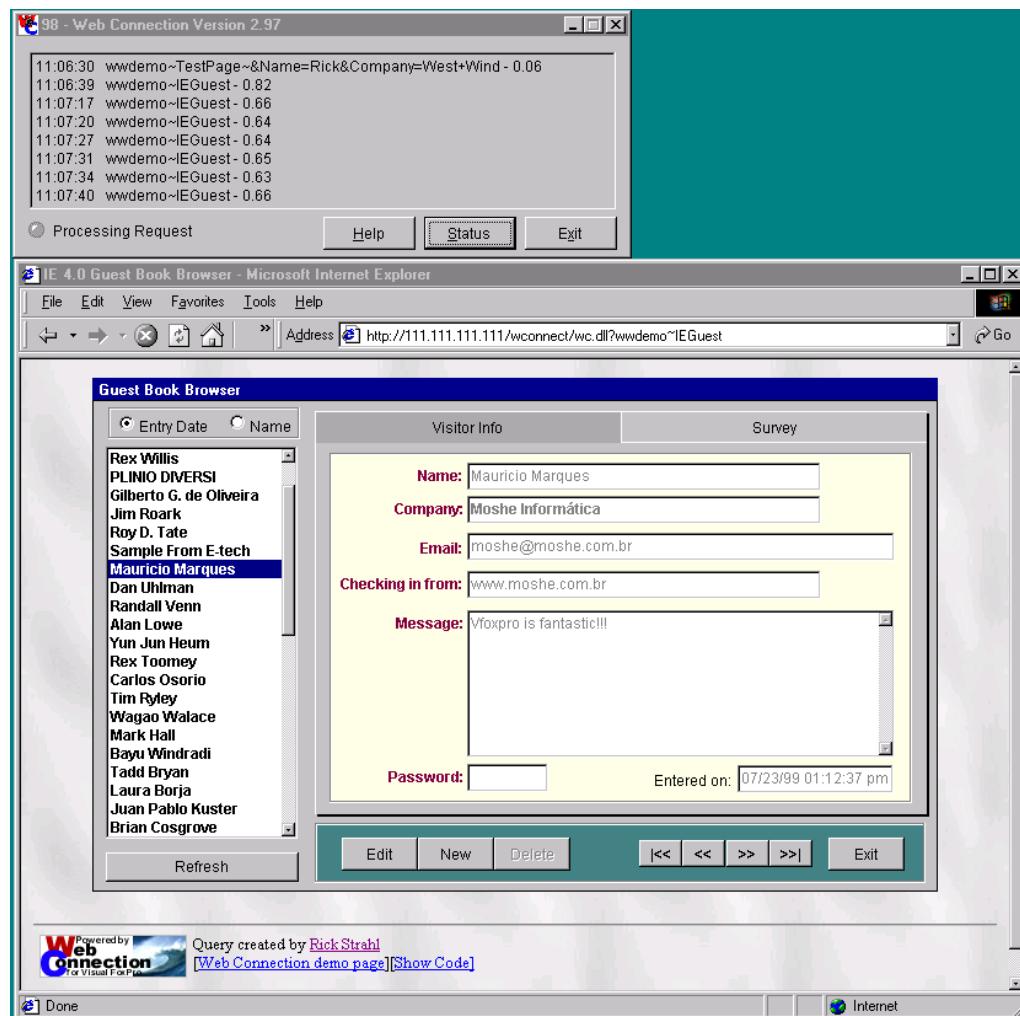


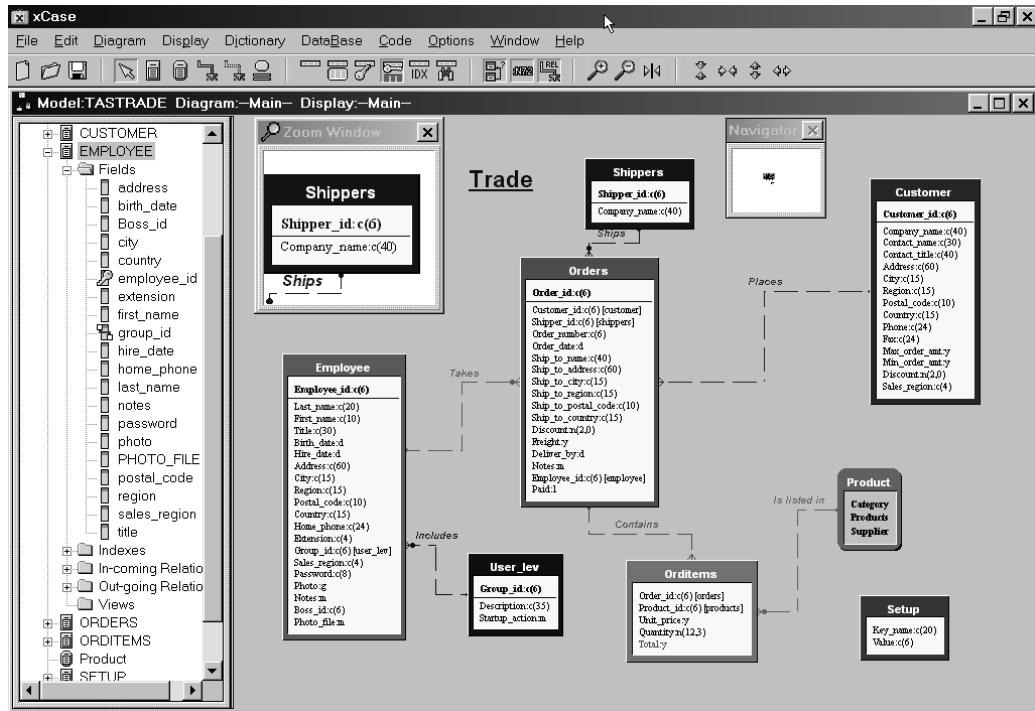
Figure 16.28. Web Connection from West Wind Technologies.

### xCase

xCase for Fox, from Resolution, Ltd., provides powerful database-design features that completely replace and greatly enhance VFP's native Database Designer. A highly sophisticated visual interface allows you to accurately design your database by capturing every detail about your customer's business information world. Then it produces high-quality diagrams, enabling you to present your design with different views and at various levels of

detail. xCase will also assist you in generating all database-related code: Data Definition Language code to build the database, triggers and stored procedures to safeguard data integrity, views and queries to extract data, and object-oriented code to provide all of the important metadata for the front end of your application. See **Figure 16.29**.

[www.xcase.com](http://www.xcase.com)



**Figure 16.29.** xCase from Resolution, Ltd.



# Chapter 17

## The Component Gallery

When a couple of good friends get together to play any type of one-on-one sport, such as tennis or racquetball, it's inevitable that a number of close shots won't quite make it—like the service return that hits the clay just outside the singles line, or the racquetball volley that just nudges the floor before hitting the front wall. In times like these, as you lay spread-eagled on the court, the one thing you don't want to hear from your opponent is the condescending admonition, "Nice try!" This is, of course, French for "You couldn't hit the ball if I held it in front of you, could you?" The first version of the Visual FoxPro Component Gallery, unfortunately, gets the same review. But before you turn the page, please read through this chapter to find out what the Component Gallery is, and why you should anxiously await the next version.

The developer tools in Visual FoxPro have come a long way since the inception of the Project Manager back in FoxPro 2.0. The PM gave us the ability to group all files in a single project in a command center, and to build executables at the push (or should that be "at the click"?) of a button.

The strength of the Project Manager—the ability to collect all files for a project into a single repository—is, when viewed from the other side, also its major shortcoming. It does not give you the ability to work with a set of components that you use to build multiple projects. In other words, as developers move to the brave new world of component-based software development, they'll need a tool to work with a workbench of components, and the Project Manager can't do this for them.

Enter the Component Gallery, new to Visual FoxPro 6.0.

Just as the Project Manager allows you to organize all components of a project in a single repository, the Component Gallery allows you to organize all of your components according to any number of criteria that you decide upon.

For example, suppose you had a number of class libraries and ActiveX controls that you regularly use in your applications—but you don't use all of them in any single application. With the Component Gallery, you could put all of your base classes, used across many applications, in a single catalog, and draw from that catalog when building a new application. Similarly, you could group your ActiveX controls into several catalogs, organized by function, and be able to easily access the control you wanted.

That's the good news. The bad news is that the interface to the Component Gallery is awkward, confusing, and behaves inconsistently, and the documentation that comes with it is so poor as to be virtually useless. I hope this chapter, in concert with the detailed discussion in Markus Egger's *Advanced Object Oriented Programming* book, will get you started properly.

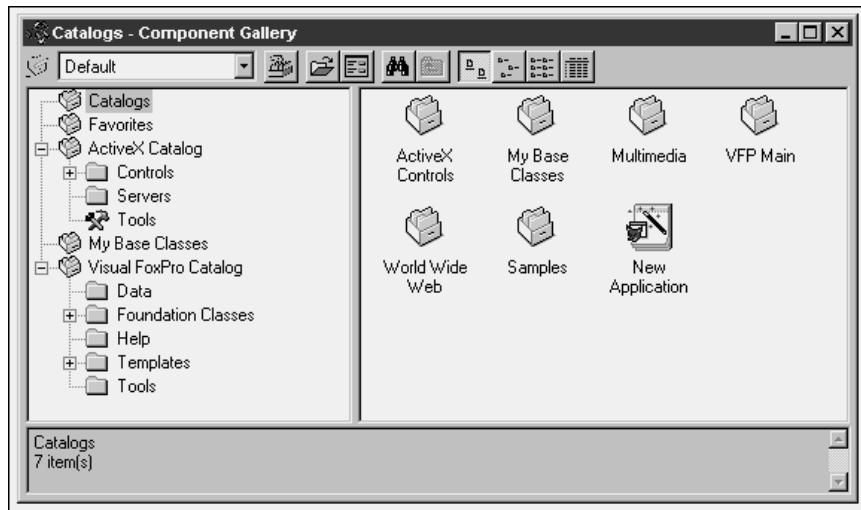
### A quick tour around the Component Gallery

As with any other tool, you first have to learn how it works by itself before you can apply it in your development work.

## Loading the Component Gallery

To load the Component Gallery, select Tools, Component Gallery, and sit back and wait for a moment or two. It's written in VFP, not C, so there's a bit of delay, depending on the resources of the machine you're using and what else you are running at the time.

The Component Gallery window has two panes. The left pane is an Explorer-style view, while the right pane shows you the items contained in the node selected in the left pane. You'll also see a toolbar with an icon, a combo box, and a number of buttons. See **Figure 17.1**.



**Figure 17.1.** The Component Gallery window has two panes in an Explorer-style view and a toolbar above them.

The leftmost icon (to the left of the combo box that says "Default") represents the selected object in the CG. You can use this icon to create new objects by dragging the icon to a form or (in the case of form classes) to the desktop.

The combo box that currently says "Default" is the View Type combo. (I'll discuss views later.) The buttons, in order from left to right, are:

- Class Browser: Switches the Component Gallery to the Class Browser. You'll most likely be keeping a lot of classes in the Component Gallery, and this button makes it handy to pop into the Class Browser when a specific class is selected in the Component Gallery.
- Open: Opens a new catalog. You can choose to get rid of all of the other catalogs (except for the Catalogs and Favorites catalogs) in the left pane, or to add the catalog you've selected to the existing Explorer list.
- Options: Opens the Component Gallery Options dialog. I'll discuss this one in detail shortly.

- Find: Allows you to find an item somewhere, anywhere, in the Component Gallery.
- Up One Level: Moves the focus of the selected item to the next higher level in the Explorer list.
- Large Icons, Small Icons, List, Details: The last four buttons on the toolbar simply change the display of the items in the right pane of the Component Gallery, just as in Windows Explorer.

## What are catalogs?

The basic premise behind the Component Gallery is that of grouping like objects into “catalogs”, and the left pane shows this. If none of the items is expanded, you’ll initially see only five nodes:

- Catalogs
- Favorites
- ActiveX Catalog
- My Base Classes
- Visual FoxPro Catalog

Because it’s an Explorer interface, a node with a (+) sign in front of it means that there are levels below it; clicking the (+) sign will expand the tree and display all the items on the next level. Similarly, a node that’s been expanded will have a (-) sign in front of it; clicking the (-) sign will close up the view.

Each of these nodes is a catalog, and in a minute I’ll explain what’s inside them, although you can make an educated guess about a couple of them as you read.

## What’s in a catalog?

In a word, shortcuts.

Just as the Project Manager contains references to forms, reports, menus, classes, and other pieces of a project, the Component Gallery simply contains references to components. The difference is that the categories in the PM are fixed, but you can group components any ol’ way you want. The object itself stays put. If you open the Properties tab of any object in the Component Gallery, you’ll see a reference to the object’s actual location, wherever it happens to reside.

Thus, it’s pretty darn important to remember that popping an item into Favorites—or any other part of the Component Gallery—does not mean that you’ve moved it. Instead, you’ve added a *link* to an existing component. Want proof? (Sure ya do—you’re a left-brainer, aren’t ya?) Shortly I’ll show you how to add components and references to components, and in that section I’ll show you that you are also creating links—not moving objects or creating duplicates of existing objects.

What this means to you right now is that you can have multiple references to the same component in different parts of the Component Gallery. For example, you could have a catalog

named “Imaging” that contains references to imaging-related components, such as ActiveX controls from third-party vendors, and classes that handle imaging functionality.

You could also have a catalog named “InterGalactic,” containing references to each of the components that you have built or that you use for applications for the InterGalactic Company. And this InterGalactic catalog could contain some of those imaging components. You don’t have multiple copies of the imaging components; instead, you have one copy of the component and several references to it.

Thus, these nodes are groupings of shortcuts. You can add your own groupings, and you can create subgroups of these as well. In fact, you can even create a grouping of other catalogs. Sorta recursive, if you know what I mean.

## Catalogs vs. folders

Just as with classes and class libraries, and files and directories, you can organize items in catalogs in different ways. For example, you could create one catalog and stuff everything into it. Though simple to understand, this solution is less than optimal when it comes to using the contents.

You could also create multiple catalogs, and group like items into the same catalog. For many people, this might be a satisfactory solution. If you need to go further, though, you can do so by using folders within catalogs. A folder in a catalog is just like a directory on your hard disk—it’s a means of grouping like items in a more granular way.

For example, if you open the Visual FoxPro Catalog, you’ll see five folders, as shown in Figure 17.1.

As another example, you already have a catalog for ActiveX controls. Within that catalog, there are a couple of folders already: one for controls and one for servers. You might delete those and create several other folders, each for a particular type of ActiveX control that you use. You could have all of your imaging controls in one folder, your data-entry controls in a second, and your form-navigation controls in a third—instead of creating three separate catalogs for them. It’s entirely up to you.

Later in this chapter I’ll discuss how catalogs and folders are stored on disk; this information might help you decide when you want to use catalogs and when to organize items in folders within a single catalog.

## The contents of the default catalogs

Before rushing off to create your own widgets, you should see what’s already out there.

### Catalogs node

You can open the items in the Catalogs node to drill down into other catalogs. Here are the three main groupings:

- ActiveX controls
- VFP stuff like foundation classes (the famous “FFC”), Help, Templates, and tools
- My Base Classes

These are discussed next.

I have some issues with the Catalogs node in the Component Gallery. First, some of the objects in the Catalogs node are also found lower in the tree. I don't understand what utility the duplication serves. Second, the navigation is awkward—you can drill down, but you can't go back up. Finally, the items that are duplicated are named slightly differently, so it's not obvious that they're the same things. As a result, I don't use the Catalogs node.

### Favorites node

What would any Microsoft tool be without a place for your favorites? If you've got zillions upon zillions of components, but you commonly use only a few of them, you can plunk them into Favorites for easy access. This is sort of like creating your own custom toolbar that has buttons for Save, Print Preview, Left Align, Right Align, Table Designer, and Run.

### ActiveX Catalog node

The ActiveX Catalog node is a bit confusing at first. It has three folders: one for controls, one for servers, and one for tools. If you drill down into the Controls folder, you'll see an item called "Installed Controls." After clicking on the item, you'll also likely see nothing in the right pane. Don't get depressed, don't go home, just right-click the Installed Controls item and select the Properties menu option. Then select the Folders tab, and notice whether the VFP Registered Controls check box is checked or not.

Then close the Properties dialog, right-click the Installed Controls item again, and select the Refresh Controls menu option. If the VFP Registered Controls check box was checked, the Component Gallery will scan through the list of controls registered on your machine and display only those that are registered with VFP, as shown in **Figure 17.2**. If the VFP Registered Controls check box was not checked, the Component Gallery will display all ActiveX controls registered on your machine. Because reading the registry is slow, refreshing the list can take some time—on a desktop P500 with 50 or 60 gigabytes of RAM, it might take about five seconds to display all 175 ActiveX controls.

You might be wondering why the Tools folder doesn't have a folder icon, but instead displays a crossed wrench and hammer. It's simply a visual effect—you can change the icon to something different (like "folder.ico") by right-clicking the Tools item, selecting the Properties menu option, and pointing the Picture value in the General tab to the icon file of your choice.

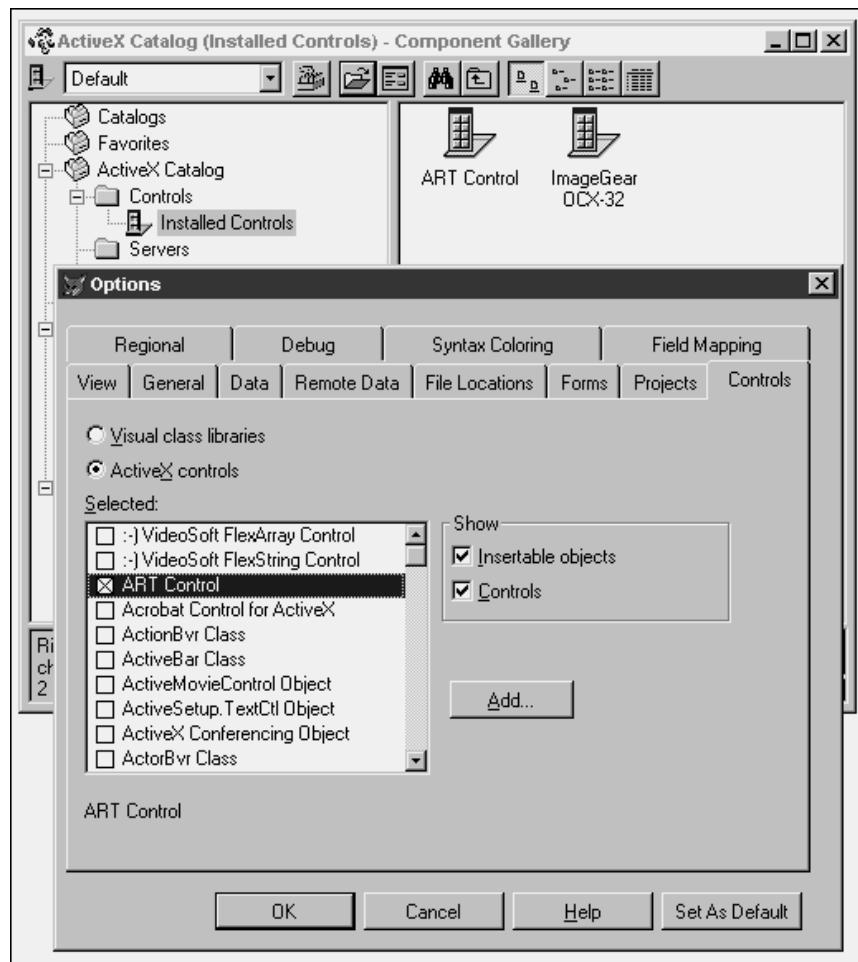
The tools inside the Tools folder include the Automation Manager (the previous version of DCOM), the DCOM Configuration Manager, Remote Automation Connection Manager, and CliReg32, a tool that allows you to register a custom VFP automation server remotely.

### My Base Classes node

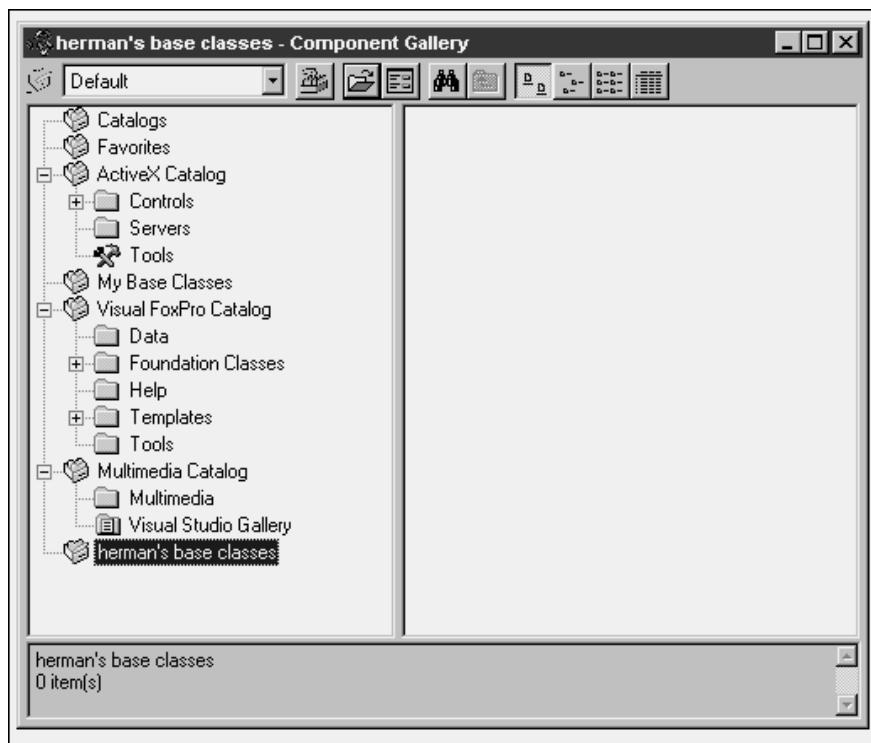
The My Base Classes node is a bit confusing, too. Aw, heck, it's all confusing, isn't it. You'd think that this might be an empty catalog where you can store references to that nifty set of base classes you've been developing since 1972. In actuality, it's already got goodies in it! The Visual FoxPro Fox Foundation Classes (FFC) have a set of base classes that you can use as your own base classes if you haven't already constructed a set. These base classes are already included in the My Base Classes catalog.

What should you do? I can't tell you what you should do, but here are some possible scenarios from which you can pick. First, you can delete the existing items in My Base Classes and add your own to replace those. Second, you could add your own base classes to the existing

set of FFC base classes. This might seem confusing, but the FFC base classes all start with an underscore. Third, you could just create your own Base Classes Catalog, as shown in **Figure 17.3**. You could rename My Base Classes to something else if it bothered you to have two similarly named catalogs.



**Figure 17.2.** The Installed Controls item expands into just those ActiveX controls that have been registered in Visual FoxPro's Tools, Options dialog.



**Figure 17.3.** You can create your own Base Classes catalog if you don't want to use the one that comes with Visual FoxPro.

### Visual FoxPro Catalog node

This has lots of treasures! Let's look.

First, open the node so you can see the five folders below it: Data, Foundation Classes, Help, Templates, and Tools.

Before I get into each of these catalogs ... well, first things first. As I mentioned earlier, the icon for ActiveX Catalog has a picture of a hammer and a wrench. But the Tools folder in the Visual FoxPro Catalog doesn't. Why not? Is it a different type of folder? Is there some mysterious functionality? A secret reason? Or could it just be a bug? In any case, if you want them both to be the same, you can right-click the Tools item, select the Properties tab, and change the Picture value in the General tab to a different image. The icon used in the ActiveX Catalog is located (on my machine) at:

```
c:\program files\Microsoft Visual Studio\vfp98\gallery\graphics\tools.ico
```

The Data folder contains a database wizard and a table wizard. The Foundation Classes folder expands to another 14 folders, each of which contains one or more classes. See Doug Hennig's articles in *FoxTalk*, December 1998 and January 1999 ([www.pinpub.com/foxtalk](http://www.pinpub.com/foxtalk)), for

excellent explanations of many of these classes. Markus Egger also has an 80-page chapter in his book, *Advanced Object Oriented Programming with VFP 6.0* (Hentzenwerke Publishing) that covers the FFC in detail.

The Help folder contains documentation for FoxTools (and a Read Me reference that is broken—it points to an .HTM file in the VFP98 directory but should be pointing to the file in the Microsoft Visual Studio directory instead).

The Templates directory expands into three more folders: one each for Applications, Forms, and Reports. Each of these directories contains a series of sample templates that is used by the appropriate wizard.

The Tools directory contains a number of useful tools, such as the Windows Calculator, the Windows Explorer, RegEdit, and, most delightfully, Filer.

### **Others**

Is that all there is? One of the major (but hidden) improvements to Visual FoxPro 6.0 over 5.0 was a huge assortment of examples and sample code. The Component Gallery is just as rich as other areas in this respect. Several other catalogs are included with VFP but don't show up immediately. Depending on which build of VFP you've installed, as well as what options you've installed, you might also have access to the Multimedia, World Wide Web, and Visual FoxPro Samples catalogs. However, they don't show up by default. I'll show you how to display all catalogs later in this chapter.

## **Where is the Component Gallery data stored?**

As you've probably figured out by now, the Component Gallery is simply a user interface to a data store. All these nodes and lists of components and shortcuts have to be stored somewhere. But where? Various parts are stored in several locations, so hang on while I lead the way through this jungle.

### **Catalog data**

First of all, each catalog is represented by a separate .DBF. When you create a catalog, you are creating a .DBF file, and each time you add a folder or an item to that catalog, the Component Gallery adds a record to that .DBF.

Catalogs that come with VFP are located in the GALLERY directory underneath VFP's home directory, while the .DBFs that you create are stored wherever you put them.



*It can be a nuisance to change over to the VFP home directory, because it's buried deep in the Program Files directory on the drive where you installed Visual Studio. The HOME() function will return the full path name where VFP is installed. Thus, the command:*

```
cd home()
```

*will change the default directory to C:\Program Files\Microsoft Visual Studio\vfp98 or wherever you installed VFP. Similarly,*

```
cd home() + "\gallery"
```

*will change to the Gallery subdirectory, and*

```
use home() + "\browser"
```

*will open the BROWSER.DBF table in the VFP root directory. As a result, you can do:*

```
cd home() + "\gallery"
dir
```

*and you'll see the listing of catalog .DBFs shown in Figure 17.4.*

| Database Table/DBF files                                           | # Records | Last Update | Size  |
|--------------------------------------------------------------------|-----------|-------------|-------|
| ACTIVEX CATALOG.DBF                                                | 12        | 09/04/1999  | 2493  |
| FAVORITES.DBF                                                      | 1         | 09/04/1999  | 1183  |
| KEYWORDS.DBF                                                       | 137       | 07/11/1999  | 4576  |
| MEDIA CATALOG.DBF                                                  | 14        | 09/04/1999  | 2731  |
| MY CLASSES CATALOG.DBF                                             | 1         | 09/04/1999  | 1184  |
| SAMPLES CATALOG.DBF                                                | 34        | 09/04/1999  | 5111  |
| VFP CATALOG.DBF                                                    | 154       | 09/04/1999  | 19398 |
| VFPGLRY.DBF                                                        | 29        | 09/04/1999  | 4516  |
| WWW CATALOG.DBF                                                    | 183       | 09/04/1999  | 13321 |
| <br>54505 bytes in 9 files.<br>476250112 bytes remaining on drive. |           |             |       |

**Figure 17.4.** The .DBF files in the Gallery subdirectory.

But there is another hierarchy of data as well—the list of catalogs. Kind of like metadata for catalogs. The file BROWSER.DBF, located in the VFP root directory, contains records for each catalog.

When you start the Component Gallery, you'll get the nodes in the following table. The table lists the names of the .DBFs.

| Node            | DBF                    |
|-----------------|------------------------|
| Catalogs        | VFPGLRY.DBF            |
| Favorites       | FAVORITES.DBF          |
| ActiveXControls | ACTIVEX CATALOG.DBF    |
| My Base Classes | MY CLASSES CATALOG.DBF |
| VFP Catalog     | VFP CATALOG.DBF        |

If you inspect the contents of BROWSER.DBF, you'll find the other catalogs that ship with VFP. You can go into BROWSER.DBF and turn them on manually, or you can go into the Catalogs tab of the Component Gallery Options dialog and select the default check box for that catalog.

As I said before, the Catalogs node is recursive; the objects in it are actually just more catalogs: ActiveX controls, Web, Multimedia, and so on. These catalogs are also stored in BROWSER.DBF, and thus also show up in the Catalog tab of the Component Gallery Options dialog. If you check the default check box, they'll show up as main nodes in the left pane of the Component Gallery window. Here are the other catalogs that show up in BROWSER.DBF:

| Node                  | DBF                 |
|-----------------------|---------------------|
| Multimedia            | MEDIA CATALOG.DBF   |
| Visual FoxPro Samples | SAMPLES CATALOG.DBF |
| World Wide Web        | WWW CATALOG.DBF     |

In addition, BROWSER.DBF contains fields that store various attributes about catalogs, such as whether or not the catalog is a global catalog, if it should show up in the list of default catalogs in the Component Gallery, and so on. For example, right-click on a catalog node and select the Properties menu option. Each of the values in the Properties dialog is stored in the catalog's record in BROWSER.DBF.

## Global settings

Global settings of the Component Gallery are also stored in BROWSER.DBF. A single record in BROWSER.DBF has an ID of BROWSER instead of FORMINFO. Each of the items in the Standard tab of the Component Gallery Options dialog, such as Enable Item Renaming and Advanced Editing Enabled, are stored in the Properties memo field of the BROWSER record in BROWSER.DBF:

```
this.lRunFileDefault = .T.  
this.lAddFileDialog = .F.  
this.lAutoLabelEdit = .T.  
this.lAdvancedEditing = .T.  
this.lFFCBuilderLock = .T.  
this.lDragDropToDesktop = .T.
```

## How to configure the Component Gallery so data isn't on C

Isn't it a joyous feeling to spend days customizing your environment, only to have Windows go belly-up on you, and make you repeat those steps? If you've been watching carefully, you've seen that all of three items in the Component Gallery have links to objects that are buried deep in the bowels of drive C. For example, one of the base classes that makes up the Fox Foundation Classes is located in:

```
c:\program files\microsoft visual studio\vfp98\ffc\_table2.vcx
```

Kind of sends a chill down your spine, eh? Do you *really* want to keep all of your catalog data stored on C? Haven't I yelled at you enough each time you wanted to do that? Hell, yes! The answer, of course, would be to move your catalog tables, as well as BROWSER.DBF, to another drive, along with the rest of your data.

However, if you've tried to find out how to point to a different drive, you've probably come up short. The reason: There isn't a specific setting for Component Gallery or BROWSER.DBF settings. Instead, you need to change the location of the Class Browser,

BROWSER.APP, in Tools, Options. Then, when the Component Gallery is started up, it will look in the same directory for BROWSER.DBF. You can't simply delete or move BROWSER.DBF, hoping that VFP will get confused and prompt you for the location, because it won't. Instead, Fox will create a new copy of BROWSER.DBF in the same directory in which it finds BROWSER.APP.

## Arranging components

So now that you know what comes "out of the box," how do you arrange the content of the box to suit your style?

### Navigating around

While it's obvious by now how to maneuver through the left and right panes of the Component Gallery, you might run into unexpected behavior occasionally, so it's worth the time to go over a few special features. One example is when the left pane of the Component Gallery appears to change contents all of a sudden, and there doesn't seem to be any way of going back except for getting out of Visual FoxPro and restarting.

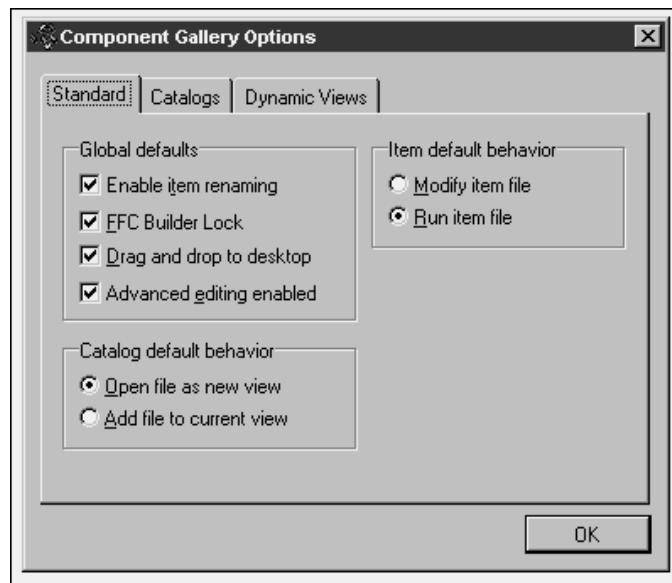
Click the Catalogs node in the left pane, so that the other catalogs are displayed in the right pane. Select one of those catalogs, such as ActiveX Controls, and right-click on it. You'll get a context menu with two menu options: Add Catalog and Open Catalog. Which of these menu options is bold (which in turn drives the behavior you get when you double-click on the catalog icon in question) depends on the setting of the Catalog Default Behavior option group in the Component Gallery Options as shown in **Figure 17.5**.

If you choose the Open Catalog menu option, all of the other catalogs (except the Catalogs and Favorites nodes) will be removed from the left pane, and the catalog you chose will be displayed along with any catalogs that are marked as Global in the Catalogs tab of the Component Gallery Options. See **Figure 17.6** for an example of the ActiveX Catalog as the only catalog in the left pane.

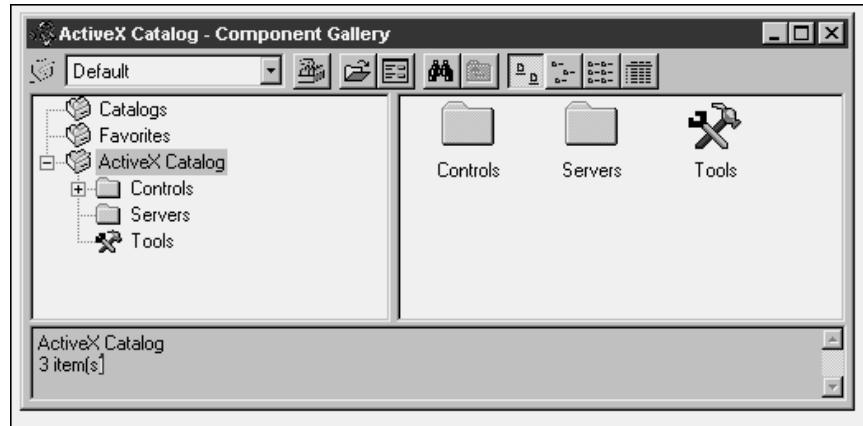
If, on the other hand, you choose the Add Catalog menu option, it will simply be added to the existing list. (If you try to add the same catalog more than once, VFP will ignore you.)

Once you have added a catalog to the left pane, you can remove it by right-clicking the catalog's node in the left pane and selecting the Remove menu option. You will be prompted to confirm your action before the catalog is removed from the list. Note that "Remove" simply removes the catalog from the list—not from your hard disk.

If you are wondering how to get back to the original list in the left pane after you've added and removed catalogs, it's easy. Close the Component Gallery and open it again. If you're wondering how to get back to the original list without closing the Component Gallery, keep wondering. You can't.



**Figure 17.5.** The Catalog Default Behavior option button in the Component Gallery Options dialog controls the default behavior of a catalog in the Component Gallery.

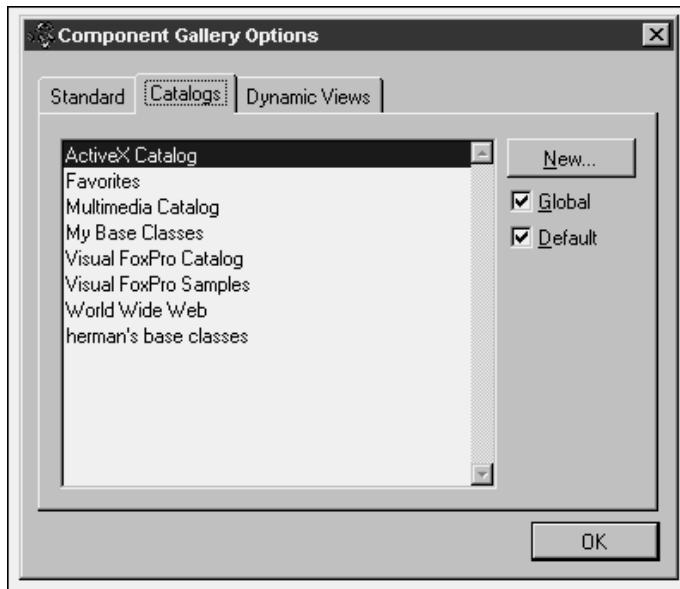


**Figure 17.6.** The ActiveX Catalog is added to the Catalogs and Favorites node through an Open Catalog menu option.

## Displaying existing catalogs that don't appear by default

As I mentioned earlier, there are a bunch of catalogs that ship with VFP that might not be displayed in the initial list. These include WWW and Samples. You can drill through the Catalogs node if you like, but that might not be satisfactory. This method also doesn't address the fact that your own catalogs won't show up in the Catalogs catalog to begin with.

The Catalogs tab of the Component Gallery Options dialog, as shown in **Figure 17.7**, allows you to select which catalogs will automatically display in the left pane of the Component Gallery when you start it up. Scroll through the list of catalogs, and notice that the Default check box is checked for the catalogs that you ordinarily see in the Component Gallery when it's opened.



**Figure 17.7.** Select a catalog and then check the Default check box to have that catalog automatically displayed when the Component Gallery is next opened.

You might be wondering how catalogs appear in the Catalogs tab to begin with. How does VFP know what catalogs are available? I'll explain the details of where all this data is stored in a later section. The short answer is that if you create your own catalog (I'll show you how to do that in the next section), it will automatically be added.

What if you have a catalog in the Catalogs catalog that doesn't show up in the Catalogs tab of the Component Gallery Options dialog? Just right-click the catalog in the right pane when Catalogs is selected in the left pane, and select Add Catalog. The next time you open the Component Gallery Options dialog, that catalog will be in the list in the Catalogs tab.

## **Creating a new catalog**

If you're going to make maximum use of the Component Gallery, you'll want to create your own catalogs. Here's how:

1. Click the Component Gallery Options button to open the Component Gallery Options dialog.
2. Select the Catalogs tab.
3. Click the New button. You will be prompted for the location and name of the .DBF that will hold the catalog.

Note that the default directory for the catalog is the current VFP default directory—which might not be the same as the vfp98\gallery directory.
4. Once you enter a name and click Save, four things happen:
  - You'll get a message that states “Catalog xxx successfully created.”
  - A new node in the Component Gallery Explorer is created.
  - A .DBF file with the name of the catalog you selected is created.
  - A new record is added to BROWSER.DBF.

Note that your catalog will be displayed in the list box in the Catalogs tab with the full path. The next time you open the Component Gallery Options dialog and select the Catalogs tab, the path will be gone and just the name of the catalog will be displayed.

## **Adding a new folder to a catalog**

You'll also want to add folders to an existing catalog. Here's how:

1. Select a catalog's node in the left pane of the Component Gallery.
2. Right-click in the right pane and select New Item, Folder. You'll see a New Folder node under the node of the catalog you selected.
3. Right-click on the New Folder name and select Rename.
4. Rename the folder to whatever you want.

## **Adding your own components**

You'll want to be able to add your own components—classes, controls, and other things—to an existing catalog. Here's how:

1. Select the folder in which you want a reference to your component.
2. Right-click in an empty space in the right pane.
3. Select New Item.
4. Select the type of component you want to add.

5. The File, Open dialog will appear, with the appropriate type of file selected. The menu options and the associated file types are shown below.

| Menu option     | File type                        |
|-----------------|----------------------------------|
| ActiveX control | Application (.EXE/.DLL/.OCX)     |
| Class           | .VCX/.VCT                        |
| Datasource      | Table/.DBF                       |
| File            | All files                        |
| Form            | .SCX/.SCT                        |
| Image           | Bitmap, Icon, .GIF, .JPG         |
| Menu            | .MNX/.MNT                        |
| Program         | .PRG                             |
| Project         | .PJX/.PJT                        |
| Report          | .FRX/.FRT                        |
| Sample          | VFP Application (.PRG/.APP/.EXE) |
| Website         | URL                              |

6. Select the file you want to add, and click OK. The component will be added to the right pane with the appropriate icon.

You will probably want to right-click on your component and add more information, such as a more robust description.

## Making a copy of a component in your own catalog

Adding a component to your own catalog really involves no new concepts. First, you'll create your own catalog, as I described above. Then you'll add the component to your catalog. Remember that when you do so, you're simply adding a reference to the component—the component stays where it was.

## Views: Finding things in catalogs

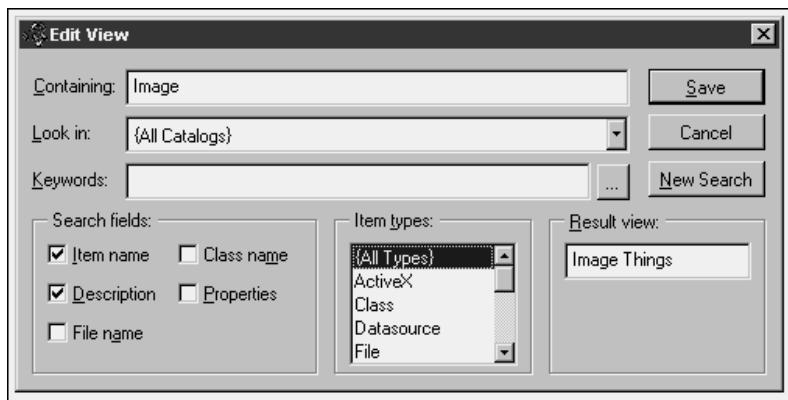
You can search one or more catalogs by several criteria. For example, suppose you wanted to find all command buttons or all controls having to do with "images." You can search through your catalogs for a word or expression, such as "Button" or "Imaging," using one or more of many criteria. But that's not all! Once you've found the items you're looking for, you can save the results of that search and refer to the results again and again. The results, unfortunately, are stored in a container called a "view"—perhaps the most overloaded word in Visual FoxPro. Nonetheless, despite this small transgression, views are quite handy.

To create a view, open the Component Gallery Options dialog and select the Dynamic Views tab. Click the New button to open the Edit View dialog, as shown in **Figure 17.8**.

You can look for a specific word or expression by entering it in the Containing text box, specify which catalog or catalogs to look in with the Look in combo box, and select one or more keywords to search on by entering them in the Keywords text box. (You can also pick predefined keywords with the ellipsis button to the right of the Keywords text box.)

You can select what parts of the catalog metadata you want to search in: Item name, Description, File name, Class name, and Properties description.

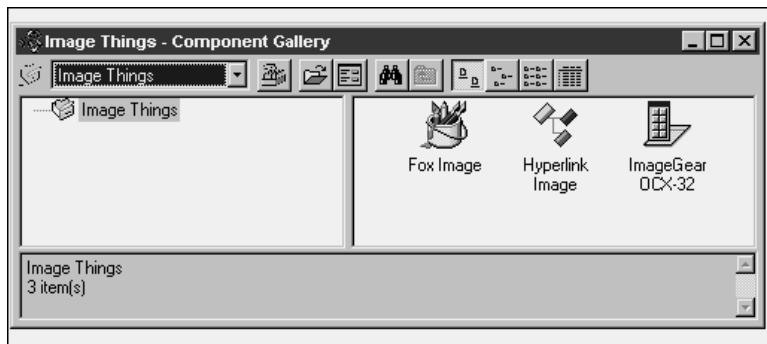
Finally, you can choose what type of item—from ActiveX control to Form to Report to Web Site—with the Item types list box.



**Figure 17.8.** The Edit View dialog allows you to search across catalogs for specific items.

Once you've chosen all of your criteria, you can name the Result view and then click Save. Here comes another confusing part. Once you click Save, you've actually done two things: You've created a view with the name you just entered in the Result view text box, but you've also executed the search.

The first time I tried using this feature, I must have created a dozen views before I realized that I wasn't going to get a list of results, or any feedback whatsoever regarding the search I did. In order to see the results of the search, you have to go back to the Component Gallery window and pick the name of the view, as shown in **Figure 17.9**.



**Figure 17.9.** To see the results of a search, select the name of the view in the View combo box in the Component Gallery window.

## Using a component in your project

Back when I wrote the VFP 3.0 book, I suggested that a 17-inch monitor was a necessity. I'm now thinking that one of those 35-inch (yes, THIRTY-FIVE-INCH) monitors would be an

appropriate choice. During development, you're going to want your PM open, of course, and space for your Form Designer or Class Designer, and the Code window and the Properties window. And you're going to want your Component Gallery open as well—cuz that's “from whence all good things shall come” from now on.

Instead of drilling down into a class library from the Classes tab in the PM, you'll grab your class from the appropriate node in the Component Gallery. That means, of course, that you'll need it open. How, exactly, do you grab items from the Component Gallery for use in various parts of your project?

Suppose you wanted to include controls and a non-visual class on a form. First, you'd create the form from the appropriate form class by choosing the form class, right-clicking, and selecting the Create Form menu option. Then, with the form visible, open the Component Gallery. (What? You mean you don't already have it opened?) Then, select the item (a class or an ActiveX control, for example) and drag it to your form. An instance of the item will be added to the form. Suppose you want to add a non-visual class to a form. You do it the same way: Find the class in the Component Gallery and drag an instance onto the form. That's all there is to it.

What else? Here are some of the things that the Component Gallery can do for you much easier than if you tried to do them without it:

- It's easier to add files to a project than by clicking the Add button in the Project Manager. With the Project Manager, you might have to navigate through a lot of directories to find the file you want; and you might not even know which .VCX contains a class you want. With the Component Gallery, you simply find the class or file you want and either drag it to the Project Manager or choose Add to Project from the shortcut menu.
- It's easier to create forms of the desired class than by changing the Form Template setting in the Forms page of Tools, Options every time you want to use a new class. Simply find the form class you want and either drag it to the Project Manager or choose Add to Project from the shortcut menu. A dialog will pop up, asking if you want to add the class (that is, the .VCX the class is in) to the project, create a new class from the selected class, or create a new form from the class.
- It's easier to add controls to a form than with the Form Controls toolbar. Unless you're adding a VFP base class to a form (as Arnold would say, “Big mistake”), you have to click the View Classes button in the Form Controls toolbar, navigate to find the .VCX (“what was the name of that .VCX that contained that control I wanted, anyway?”), then figure out which of the seven identical buttons in the toolbar represents the class you want, and finally drag it to the form. With the Component Gallery, you simply find the class you want and either drag it to the form or choose Add to Form from the shortcut menu.
- It's easier to subclass an existing class. As I mentioned earlier, even if a class is selected in the Project Manager, the New Class dialog displayed by clicking the New button doesn't automatically choose that class for the Based On class. With the Component Gallery, you simply find the class you want and either drag it to the

Project Manager or choose Add to Project from the shortcut menu, and then choose the Create New Class option from the dialog that appears.

- It's easier to test a class. Just like with the Class Browser, you can "run" a class by dragging it to the VFP screen; you can also choose Run from the shortcut menu. In the case of form classes, a form is instantiated from the class and a public variable called oForm<n> contains a reference to it, so you can easily access or change properties or call methods. You can also view or run a sample of a class if one is defined by choosing the appropriate option from the shortcut menu.

## **Summary**

The more I worked with the Component Gallery, the more I liked it—and hated it at the same time. The idea of a centralized source for all of the pieces that you put into an application really appealed to me; indeed, it felt more like a Component "Library" than a "Gallery." I mean, this is *great!* But at the same time, there are problems. Always problems. First, it's sorta pokey to load—even on a really fast machine with lots of resources. And this is a double problem because of the way that the Component Gallery is loaded. Many developers issue the CLEAR ALL command about once every three minutes—and that closes the Gallery because it's an object in memory. So then you have to open it again, which takes another three minutes, by which time you've forgotten what problem caused you to issue CLEAR ALL in the first place.

Also, there are enough quirks in the interface and feature set that I feel it's too much trouble for a single developer to use unless they're going to make a concerted effort. And the documentation on how to use it is abysmal. (Check out Ken Levy's video on the Component Gallery at [www.classx.com](http://www.classx.com) for more in-depth info on using it by one of the masters.) On the other hand, however, significant benefits can accrue for a team of developers. Having a single repository of components available to every developer is truly a "killer app," and I know of at least one large team who has successfully used the Component Gallery during the development of a very large application.

I'm looking forward to seeing enhancements and fixes in the Component Gallery for VFP 7.0.

# Chapter 18

## Project Hooks

For the most part, the design surfaces that Visual FoxPro makes available to a developer provide fixed functionality—you get what you get, and nothing more. If you want additional functionality to, say, the Form Designer, you have to hack the .SCX file produced by the Form Designer—there is no way to hook into the Form Designer and add your own mechanisms (outside of builders). The Class Browser and Component Gallery have broken the mold, allowing a developer to get into the design surface, and, more importantly, to add their own programs to enhance or just simply change the default behavior. Visual FoxPro 6.0 extends this functionality to one of the original tools of FoxPro—the Project Manager—via a pair of mechanisms called Project Objects and Project Hooks. In this chapter I'll discuss each one, explain how they work, and show what you might use them for.

The Project Manager stores its information in a pair of files, the .PJX and .PJT—a table and associated memo file. If you wanted to extend or add functionality to the Project Manager, you had to open the .PJX as a table and manipulate the information inside, just as you did any other database. Developers often did this because there were a whole raft of things they wanted to do with some or all of the files that made up a project.

Visual FoxPro 6.0's Project Manager has two new features. When a project is opened, a project “object” is created. This object contains events, methods, and properties just like any other object you've seen in VFP. You can work with these PEMs just as any other object, thus negating the requirement to open the .PJX as a table.

You can also add your own behaviors to the Project Manager, via a Project Hook class that you specify through the Projects tab of the Tools, Options dialog or the Project tab of the Project, Project Info dialog, just as you can with the Class Browser and Component Gallery.

### Accessing the project object

If you decided to blow off the rest of this chapter, thinking you could just spelunk around for the project object, you probably started VFP, opened a project, and then did a DISPLAY MEMORY, looking for an “oProject” variable to show up alongside \_oBrowser.

And it didn't work, so now you're back to reading the damn book, right? Don't feel bad; I did the same thing after I heard someone mention the new “project object.” In fact, it's a bit more complicated, and requires explanation of a topic I've not covered yet in this book.

### Application object

When you start Visual FoxPro, an “application object” is created automatically. This object exposes a set of properties, events, and methods that can be accessed within VFP (and, interestingly enough, from outside VFP as well—see the Note below).



*One of the beauties of Windows is the ability of applications to talk to each other. This is done through a mechanism called Automation, which began as Dynamic Data Exchange (DDE)—allowing applications to exchange data—and then became Object Linking and Embedding (OLE)—allowing applications to directly talk to each other's data and automatically call the originating program. This finally spawned OLE Automation, where applications could reference each other's properties and methods in programs. OLE Automation was then renamed Automation because the term “OLE Automation” was perceived as being proprietary. (Well, it was. Duh!) Automation was renamed COM and the interface was opened up, and the latest player in this scenario is the “new, improved COM”—COM+. But I digress.*

*An application such as Visual FoxPro (or Excel, or Word) that participates in Automation can be called from another application, and commands and messages can be passed back and forth. For example, Visual FoxPro can open a copy of Excel, pass some data to Excel, tell that copy of Excel to perform a calculation on that data, and have Excel pass the resulting answer back to VFP. This copy of Excel is actually an “instance,” created through a CREATEOBJECT command. Visual FoxPro can be treated in the same manner; for example, the instance of VFP that Word might create is this same application object.*

The application object has properties such as ActiveForm (an object reference to the active form on the desktop if there is one), Caption (the text displayed in the VFP desktop’s title bar), FullName (the file name of an instance of VFP and the path to the directory from which the instance was started), StatusBar (the text displayed in the status bar of the instance of VFP), StartMode (defines how VFP was started—interactively, as an .APP or .EXE, or an in-process or out-of-process server), and VersionNumber (the version number of the instance of VFP).

You can use the following commands to access these properties from within the Command window:

```
? application.caption  
Microsoft Visual FoxPro  
application.caption = application.caption + " (Herman rules!)"  
? application.caption  
Microsoft Visual FoxPro (Herman Rules!)  
? application.version  
6.0  
? application.StartMode  
0
```

This is pretty long, so you can also use the \_VFP system memory variable for the same purpose:

```
? _vfp.caption
Microsoft Visual FoxPro
_vfp.caption = _vfp.caption + " (Herman rules!)"
? _vfp.caption
Microsoft Visual FoxPro (Herman Rules!)
? _vfp.version
6.0
? _vfp.StartMode
0
```

Some methods of the application object include DoCmd(), Eval(), Quit(), and Help(). DoCmd() and Eval(), for instance, allow another application to run a copy of VFP and execute a VFP command or evaluate an expression remotely, respectively.

## Projects collection

Two more properties of the application object are ActiveProject and Projects—and those are the ones of interest today.

However, Projects is not a property like the others I just described. Instead, Projects is an array property that contains a list of all open projects, with the most recently opened project at index 1, and the first project opened at the end of the array. (This list of open projects is called a *collection*; there are other collections in VFP as well, such as the collection of open forms.)

Remember how I said that a project object is automatically created when a project is opened? The Projects collection actually references each of these project objects—so you can drill down from the application object to a project object to specific properties of an individual project.

The ActiveProject is simply a pointer to whichever project in the Projects collection is “on top.” Thus, it too is a reference to a project object.

To sum up, when an instance of Visual FoxPro is created, a VFP application object is also created. This application object contains a number of collections, one of which is a collection of open projects. So how about a look at project objects?

## Project object

Just as you can run a form and access the properties, events, and methods of that form, once you have a project open, you can access the properties, events, and methods of its project object. Unfortunately, while you can determine the PEMs of a form by opening the Properties window when the form is open, the same Properties window doesn’t open when you open a project in the Project Manager. Instead, you have to look in the online help (under Project Object). **Table 18.1** lists the properties of the project object that are specific to it.

Because each of these is a property of a project object, you’ll need to reference which project you want information from. For example, suppose you have one project open. You can use either of the following commands in the Command window to get the fully qualified name of the project:

```
? application.activeproject.name
f:\InterGalactic\BooksInventory\source\it.pjx
? application.projects[1].name
f:\InterGalactic\BooksInventory\source\it.pjx
```

**Table 18.1.** Properties of a project object.

| <b>Property</b>    | <b>Access</b> | <b>Type</b> | <b>Description</b>                                                                                                                            |
|--------------------|---------------|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| Autolncrement      | R/W           | L           | Specifies if the build version number of a project should be automatically incremented each time the .EXE or .DLL is built.                   |
| BuildDateTime      | R/O           | T           | The date and time of the last build.                                                                                                          |
| Debug              | R/W           | L           | Specifies if debugging information is included with compiled source code.                                                                     |
| Encrypted          | R/W           | L           | Specifies if files are encrypted as they're compiled.                                                                                         |
| Files              | R/O           | Coll        | A collection of file objects.                                                                                                                 |
| HomeDir            | R/W           | C           | The home directory of the project.                                                                                                            |
| Icon               | R/W           | C           | Specifies the icon for a form when the form is minimized, and the icon displayed for a distributed .EXE.                                      |
| MainClass          | R/O           | C           | The name of the ActiveDoc class used as the main program (used only when MainFile contains the name of the .VCX where this class is defined). |
| MainFile           | R/O           | C           | The main program in the project, set using the SetMain() method.                                                                              |
| Name               | R/O           | C           | The fully qualified path and file name of the project.                                                                                        |
| ProjectHook        | R/W           | O           | A reference to the project hook object associated with the project.                                                                           |
| ProjectHookClass   | R/W           | C           | The name of the class the project hook object is instantiated from.                                                                           |
| ProjectHookLibrary | R/W           | C           | The class library the ProjectHook class is defined in.                                                                                        |
| SCCPProvider       | R/O           | C           | The source code control provider for a project.                                                                                               |
| ServerHelpFile     | R/W           | C           | The help file for the typelib used for server classes.                                                                                        |
| ServerProject      | R/W           | C           | The first part of the ProgID for the Automation server created by the project; the default is the same as the Name property.                  |
| Servers            | R/O           | Coll        | A collection of server objects.                                                                                                               |
| TypeLibCLSID       | R/O           | C           | The typelib Registry CLSID (Class Identifier) for a type library created for server classes in a project.                                     |
| TypeLibDesc        | R/W           | C           | The description for a type library created for server classes in a project.                                                                   |
| TypeLibName        | R/O           | C           | The fully qualified path to the typelib for the project.                                                                                      |
| VersionComments    | R/W           | C           | The comments for a project.                                                                                                                   |
| VersionCompany     | R/W           | C           | The company name information for a project.                                                                                                   |
| VersionCopyright   | R/W           | C           | The copyright information for a project.                                                                                                      |
| VersionDescription | R/W           | C           | The description for a project.                                                                                                                |
| VersionLanguage    | R/O           | C           | The language ID for a project.                                                                                                                |
| VersionNumber      | R/W           | C           | The build number for a project in the format MMMM.mmmm.bbbb.                                                                                  |
| VersionProduct     | R/W           | C           | The product name for a project.                                                                                                               |
| VersionTrademarks  | R/W           | C           | The trademark information for a project.                                                                                                      |
| Visible            | R/W           | L           | Specifies whether or not to display the Project Manager.                                                                                      |

However, suppose you opened projects A, B, and C (in that order). ActiveProject would return the name of the last project you opened (C.PJX), because it was the active project. You could also reference any of the projects like so:

```
? application.projects[1].name
f:\InterGalactic\BooksInventory\source\C.pjx
? application.projects[2].name
f:\InterGalactic\BooksInventory\source\B.pjx
? application.projects[3].name
f:\InterGalactic\BooksInventory\source\A.pjx
```

Notice that the first project that was opened is at the end of the list.

A project object also has methods, and you can probably guess what some of these are. See **Table 18.2**.

**Table 18.2.** Methods of a project object.

| Method    | Description                                                                                |
|-----------|--------------------------------------------------------------------------------------------|
| Build()   | Rebuilds a project or creates an .APP, .EXE, or .DLL.                                      |
| CleanUp() | Cleans up a project table by removing records marked for deletion and packing memo fields. |
| Close()   | Closes a project and releases the project's ProjectHook and Project objects.               |
| Refresh() | Refreshes a project's visual display.                                                      |
| SetMain() | Sets the main file in a project.                                                           |

## Files collection

Just like the application object has a Projects property that is a collection of projects open in the application, a project object has a Files property that is a collection of files in the project, a Count property that identifies how many files are in it, and the Add() method that allows you to add an item to the project.

## File object

Similarly, the Files collection is actually a collection of file objects. **Table 18.3** lists the properties of a file object.

Because a file object is contained in a project object, the syntax becomes a little long. To get the Type for a File, you'd use the command:

```
? application.activeproject.files[1].type
```

And, of course, a file object also has methods. See **Table 18.4**.

## Servers collection

Using Visual FoxPro, you can build a special type of application called an Automation Server. These applications are called from another Windows application, such as Excel. For example, you could build an Automation Server in VFP that would look up a postal code in a table and return the city and state or province for that postal code. Then, you could call that server from Excel to do a lookup of a city based on a postal code.

You can have one or more Automation Servers in a project, and they have special properties. Thus, along with the Files collection, a project object also has a Servers collection and a Count property.

## Server object

By now, it probably goes without saying that the Servers collection is actually a collection of server objects. **Table 18.5** lists the properties of a server object.

The syntax to address a server object is similar to that of a file object. To get the ProgID for a server, you'd use this command:

```
? application.activeproject.servers[1].ProgID
```

Unlike project and file objects, though, a server object does not have any methods.

**Table 18.3.** Properties of a file object.

| Property         | Access | Type | Description                                                                                           |
|------------------|--------|------|-------------------------------------------------------------------------------------------------------|
| CodePage         | R/O    | N    | The code page for the file.                                                                           |
| Description      | R/W    | C    | The description for the file or server class.                                                         |
| Exclude          | R/W    | L    | Specifies if a file is excluded from an application, executable, or .DLL.                             |
| FileClass        | R/O    | C    | The class a form is based on.                                                                         |
| FileClassLibrary | R/O    | C    | The library the form's class is defined in.                                                           |
| LastModified     | R/O    | T    | The date and time the file was last modified.                                                         |
| Name             | R/O    | C    | Fully qualified path and file name.                                                                   |
| ReadOnly         | R/O    | L    | Specifies whether the user can edit the file.                                                         |
| SCCStatus        | R/O    | N    | A numeric value indicating the source code control status of the file.                                |
| Type             | R/O    | C    | A character value indicating the file type, such as V for .VCX, P for program, K for form, and so on. |

**Table 18.4.** Methods of a file object.

| Method             | Description                                                                                                                   |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------|
| AddToSCC()         | Adds a file in a project to source code control.                                                                              |
| CheckIn()          | Checks in changes made to a file in a project under source code control.                                                      |
| CheckOut()         | Checks out a file in a project under source code control.                                                                     |
| GetLatestVersion() | Gets the latest version of a file in a project under source code control.                                                     |
| Modify()           | Opens a file in a project for modification.                                                                                   |
| Remove()           | Removes a file from its Files collection and project.                                                                         |
| RemoveFromSCC()    | Removes a file in a project from source code control.                                                                         |
| Run()              | Runs or previews a file in a project.                                                                                         |
| UndoCheckOut()     | Discards any changes made to a file in a project under source code control and checks the file back into source code control. |

**Table 18.5.** Properties of a server object.

| Property           | Access | Type | Description                                                                                         |
|--------------------|--------|------|-----------------------------------------------------------------------------------------------------|
| CLSID              | R/O    | C    | Contains the registered CLSID (Class Identifier) for the server.                                    |
| Description        | R/W    | C    | The description of the server class.                                                                |
| HelpContextID      | R/W    | N    | Specifies a context ID for a topic in a Help file to provide context-sensitive Help for the object. |
| Instancing         | R/W    | N    | Specifies how the server can be instantiated.                                                       |
| ProgID             | R/O    | C    | Contains the registered ProgID (Programmatic Identifier) for a server in a project.                 |
| ServerClass        | R/O    | C    | The name of a server class.                                                                         |
| ServerClassLibrary | R/O    | C    | The library the server class is defined in.                                                         |

## What next?

So by now you have a pretty good idea of the object model of the application—project—file/server hierarchy. But what good is this information? By itself, it doesn’t do you much good; you’re not likely to issue a bunch of `application.project[n].build()` commands in the Command window, are you? Probably not. However, there are two things you would want to do:

- Build programs that programmatically manipulate the project.
- Write programs that are automatically executed when things happen to the Project Manager or its contents.

## Project tools

In this section, I’m going to show you how to build a simple program that allows you to edit the descriptions of the files in a project in one step, instead of having to use the Project, Edit Description menu option for each individual file in the project. While this is nothing that I’m going to want to package and try to sell, it will serve nicely as a means of demonstrating the use of the various pieces of the project object model that I went over in the last section.

First, I’ll build a version of the routine that replaces any empty description with the phrase “Missing!”, and then displays all of the files together with their type and description in a message box. Because many readers of this book are coming into VFP from FoxPro 2.x, I’ve listed this code in both “traditional” form and with the new-to-VFP FOR EACH construct, which is preferred for working with collections. Note that this code doesn’t do any checking at all and depends on having a project open.

```
* traditional construct with "FOR I = n to m
m.lcX = ""
for m.li = 1 to application.activeproject.files.count
  if empty(application.activeproject.files[m.li].description )
    application.activeproject.files[m.li].description = "Missing!"
  endif
```

```
m.lcX ;
= m.lcX ;
+ chr(13) ;
+ application.activeproject.files[m.li].type ;
+ " " ;
+ application.activeproject.files[m.li].name ;
+ " " ;
+ application.activeproject.files[m.li].description
next
messagebox( m.lcX )

* construct demonstrating FOR EACH
m.lcX = ""
for each loFile in application.activeproject.files
  if empty(loFile.description)
    loFile.description = "Missing!"
  endif
  m.lcX ;
  = m.lcX ;
  + chr(13) ;
  + loFile.type ;
  + " " ;
  + loFile.name ;
  + " " ;
  + loFile.description
next
messagebox( m.lcX )
```

The code is fairly straightforward. First, a memory variable that will hold the string of files is initialized. Next, a FOR...NEXT loop is started, looping through all files in the active project. For each file in the project, the description is checked. If it's empty, the description is stuffed with the phrase "Missing!" Next, the type, name, and description are added to the current list of files. Finally, after the loop is done, a message box displays the final string.

Note the syntax for the Files collection—the counter is an argument of the Files collection, so each file in the collection is addressed in turn, but the item in the Files collection has multiple properties, such as type and name. Common mistakes are to forget the argument, just using syntax like this:

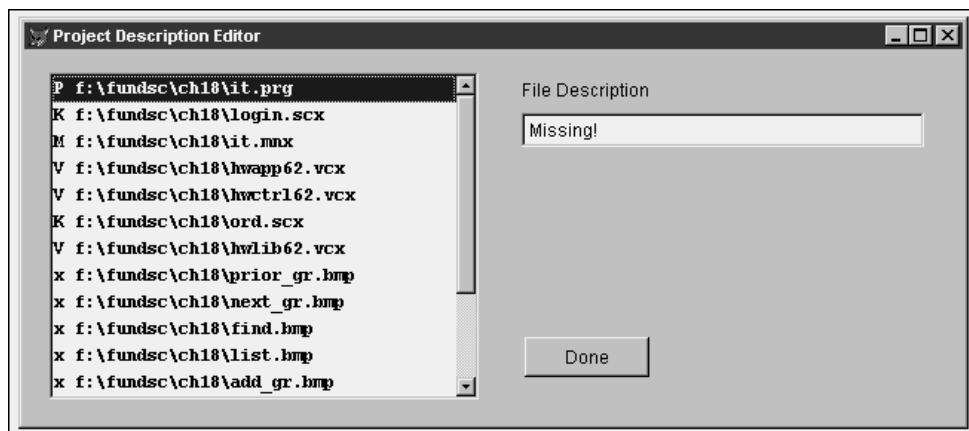
```
application.activeproject.files.type
```

or to try to append the argument to the Files collection property, like so:

```
application.activeproject.files.type[m.li]
```



So far, so good. How about dropping a user interface around this code? The form in **Figure 18.1** shows a simple interface for displaying all files in the project as well as the description of the highlighted file. The files in this section are in the CH18 directory of the source code downloads for this book.



**Figure 18.1.** The Project Description Editor.

The list box displays the type of file and the actual file name. As you scroll through the list of files, the description for the current file displays in the text box to the right of the list box. The array that supports the list box has four columns as shown in **Table 18.6**.

If you change the contents of the Description text box for a specific file, the value in the fourth column of the `hwlstFiles.aItems` array will be changed as soon as the Description text box loses focus. However, this updates just the array—not the description in the project itself. When you click the Done button, the Description values in the fourth column update the project description.

**Table 18.6.** Contents of the `hwlst.aItems` array.

| Column   | Type of Data                         |
|----------|--------------------------------------|
| Column 1 | Concatenated string of Type and Name |
| Column 2 | Type                                 |
| Column 3 | Name                                 |
| Column 4 | Description                          |

The code that fills the list box is in the `Init()` of the form:

```
with thisform.hwlstfiles
  decl .aItems[application.activeproject.files.count,4]
  for each loFile in application.activeproject.files
    .aItems[m.li,2] = loFile.type
    .aItems[m.li,3] = loFile.name
    .aItems[m.li,4] = loFile.description
    .aItems[m.li,1] = .aItems[m.li,2] + " " + .aItems[m.li,3]
  next
  .requery()
  .ListIndex = 1
endwith
```

The code that fills the text box with the description for the highlighted item in the list box is in the AnyChange() method of the list box:

```
thisform.hwtxtDesc.Value = this.aItems[this.listindex, 4]
```

The code that updates the fourth column of the array in the list box once the text box loses focus is in the LostFocus() method of the text box:

```
thisform.hwlstFiles.aItems[thisform.hwlstFiles.listindex, 4] = this.value
```

The code that updates the project description from the fourth column of the list box array is in the Done command button:

```
for m.li = 1 to application.activeproject.files.count  
  application.activeproject.files[m.li].description =  
  this.hwlstFiles.aItems[m.li,4]  
next  
this.release()
```

Fairly straightforward, isn't it? It used to be quite a chore having to open a project file as a table, grab the contents of the appropriate fields, manipulate them as you wanted, and then mess with the project file's table again to update the information. With the project object model, you have direct access to the project and its contents.

## Project hooks

In my Bonus Session at the 1998 Microsoft Visual FoxPro DevCon, I demonstrated a little program that intercepted a programmer's use of the files in a project. For example, when the user tried to open a form, they received a message, purportedly from the Project Manager that was demonstrating a mind of its own, that it wanted to open a report instead, and did so.

This particular example might not be terribly useful during day-to-day production programming work, but it shows that you can now add hooks to the Project Manager to add or replace functionality as you need.

## Available functions that can be intercepted

So what all can you do? Just about anything. Technically, you'll instantiate a "project hook object" when opening a project. (I'll describe how this works in the next section.) From then on, certain project hook events are fired when something is done to the project. The following sections describe the places where you can add your own functionality, and the event you would "hook" into in order to do so.

### At the beginning of the build process

The BeforeBuild event is fired when you click the Build button in the Project Manager, when you issue a BUILD command (such as BUILD APP or BUILD EXE), or when you call the Build() method of a project object.

You could use this to simply intercept the user before the build starts, to ask them to verify whether or not they really, really want to build the project. If they answer No, you would use the NODEFAULT command to prevent the build from being performed.

The parameters you pass to this method include items such as the name of the results file (the name of the .APP, .EXE, or .DLL you are building), the type of file (an .APP, an .EXE, or a .DLL), whether or not to rebuild all, and so on. Setting these values in the BeforeBuild event would effectively negate any options that the user has set.

### **After the build process is finished**

The AfterBuild event is fired at the end of the build process. You could log the results of the build, such as the name of the project and the date and time of completion, to a table at the completion of the build. If an error occurs during the build, the error number is passed to the AfterBuild event—you could save this as part of the saved data.

### **When a file is added to the Project Manager**

QueryAddFile is fired in a number of places. For instance, after you press the New button, save the file, and close the respective designer, QueryAddFile is executed. It is also fired after you click the Add button and select a file in the Open dialog, or when you call the Add() method of the project object's File collection. Note that the file has already been selected by the time this method is fired—one oft-mentioned enhancement request has been for a “BeforeAdd” method.

Like BeforeBuild, the NODEFAULT command will prevent the file from being added to the project.

### **When a file is modified in the Project Manager**

The QueryModifyFile event is fired when you click the Modify button in the Project Manager, or when you call the Modify() method of a file object.

It does not get fired if you simply modify the file through a MODIFY command in the Command window.

Like BeforeBuild, the NODEFAULT command will prevent the file from being modified.

### **When a file is removed from the Project Manager**

The QueryRemoveFile event is fired when you click the Remove button in the Project Manager, or when you call the Remove() method of a file object.

Like BeforeBuild, the NODEFAULT command will prevent the file from being removed from the project.

### **When a file is executed from the Project Manager**

The QueryRunFile event is fired when you click the Run button in the Project Manager, or when you call the Run() method of a file object.

Like BeforeBuild, the NODEFAULT command will prevent the file from being executed.

## **When a file is dragged over or dropped on the TreeView control in the Project Manager**

You can also add a file to the Project Manager by dragging and dropping a file, say, from Windows Explorer. There are also a couple of OLE Drag and Drop events available with respect to the Project Manager. As a result, you can add your own functionality at these places as well.

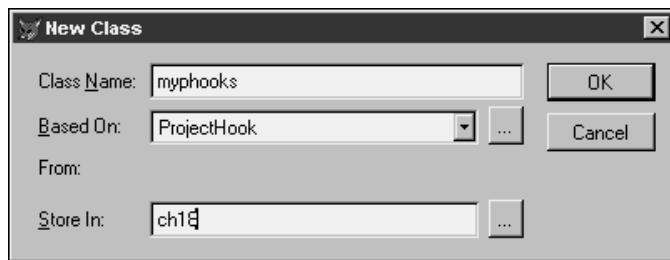
The OLEDragOver event is fired when you drag a file over the TreeView control in the Project Manager. If you let go (drop the file), the OLEDragDrop event fires. As noted earlier, the QueryAddFile event is fired after the OLEDragDrop event fires.

## **Enabling a project hook**

Now that you've seen all the places where you can hook into the Project Manager's interactions, you probably want to try this yourself. The basic idea is to create a class in a class library, based on the new VFP 6.0 ProjectHook class, and either add it to a specific project, or configure VFP to add the class to all new projects. I'll first go through the steps to add a project hook to a single project, and then discuss what's different when you want to do so globally. The source code for this section is in the CH18 directory of the downloads for this book.

### **Enabling a project hook for a specific project**

Suppose you've got a project and you want to display a message box to the user in various hooks of the Project Manager. First, create a project, naming it "IT" like always. Next, create a class named MyPHooks, add it to a class library file called CH18 (I'm losing creativity tonight), and base it on the ProjectHook class, as shown in **Figure 18.2**.



**Figure 18.2.** Create a new ProjectHook class for your project.

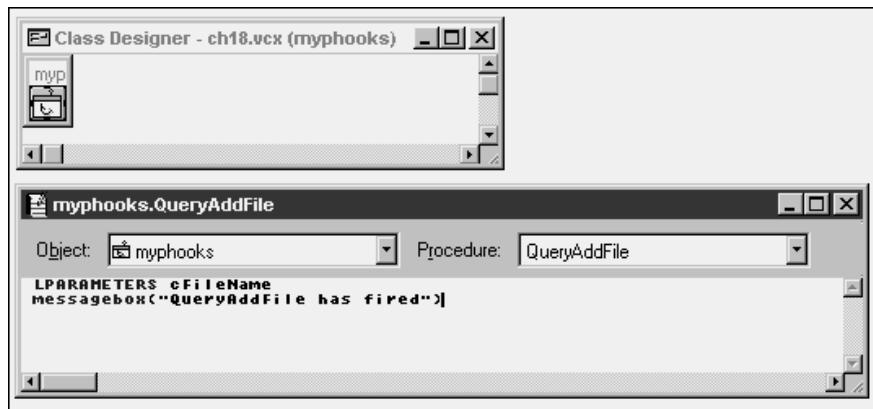
Next, open the Code window of the Class Designer, look for the QueryAddFile() method, and add a line of code, as shown in **Figure 18.3**:

```
messagebox("Add file")
```

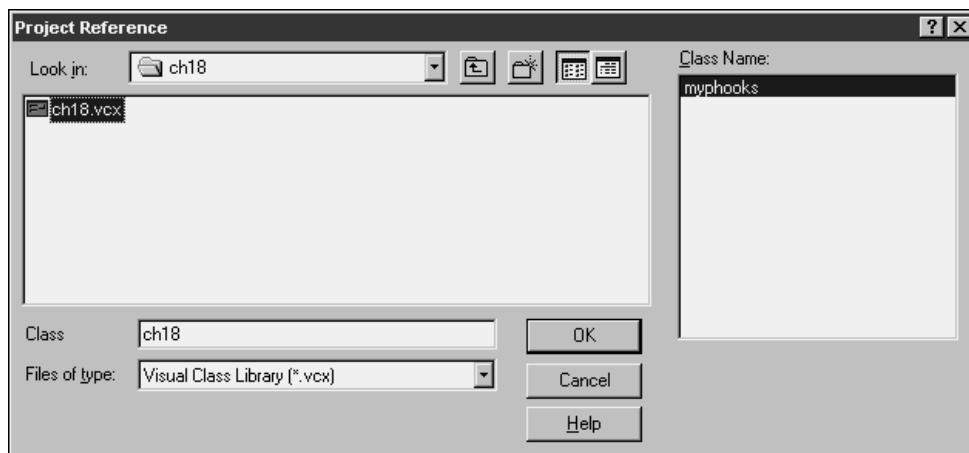
Do the same for several other events, such as BeforeBuild, AfterBuild, QueryModifyFile, and so on. Then save the class and close the Class Designer.

Next, add your new ProjectHook class to the IT project you just created. Select the Project, Project Info menu option, and select the Project tab if it's not already selected. Click the Project Class check box, and select the MYPHOOKS class in the CH18.VCX as shown in **Figure 18.4**.

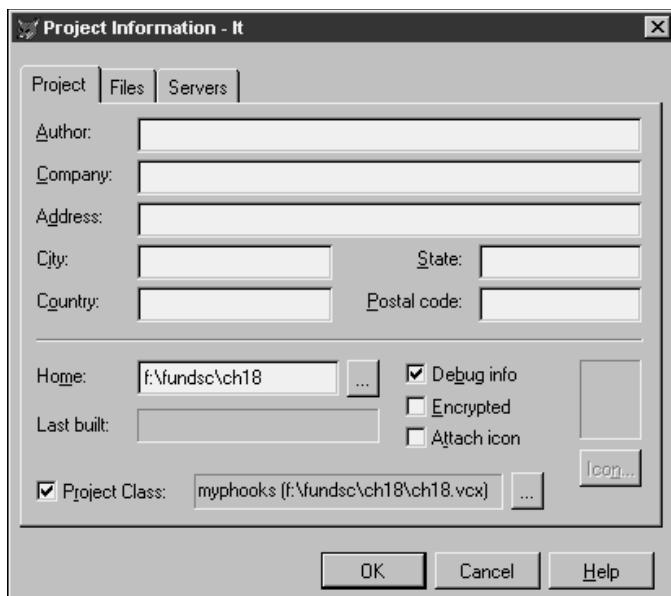
Once you click the OK button in the Project Reference dialog, the class name and class library name will appear in the read-only text box in the Project Information dialog as shown in **Figure 18.5**.



**Figure 18.3.** Add a messagebox command to the QueryAddFile() method.



**Figure 18.4.** Add the MYPHOOKS class (in the CH18.VCX class library) to the IT project.



**Figure 18.5.** The IT project with the MYPHOOKS class set as the Project Class.

Click OK in the Project Information dialog, and you'll be returned to the Project Manager.

Now here's a trick. If you try executing one of those actions, such as adding a file or building the project, you won't see any message boxes. You have to close the project and open it again for the new project hook to "take." (Underneath the hood, although you've defined the ProjectHook class, the project hook object hasn't been instantiated because it's only created when the project is opened.) Do that now: Close the window and open the project again. Then add a file, and the message box will fire as expected.

### Enabling a global project hook

You can also attach a ProjectHook class to all of your projects. After creating the ProjectHook class as described above, select the Projects tab of the Tools, Options dialog. Add the ProjectHook class as shown in **Figure 18.6**.

Now, when you create a new project, the ProjectHook class you specified in Tools, Options will automatically be placed in the new project's Project Class field in the Project tab, and you'll have access to it just as if you had added the Project Class only to that project.

Note that you'll have to specify a ProjectHook class for existing projects—setting a ProjectHook class in the Tools, Options dialog does not search for every existing VFP project on your computer!

### Getting in and around a ProjectHook setting

As I mentioned earlier, when you open a project that has a ProjectHook class attached to it, the project hook object is automatically instantiated and its methods are available for you to

manipulate. There might be times when you don't want those events to fire—you can do this by opening the project with the following command:

```
modify project MYPROJ noprojecthook
```

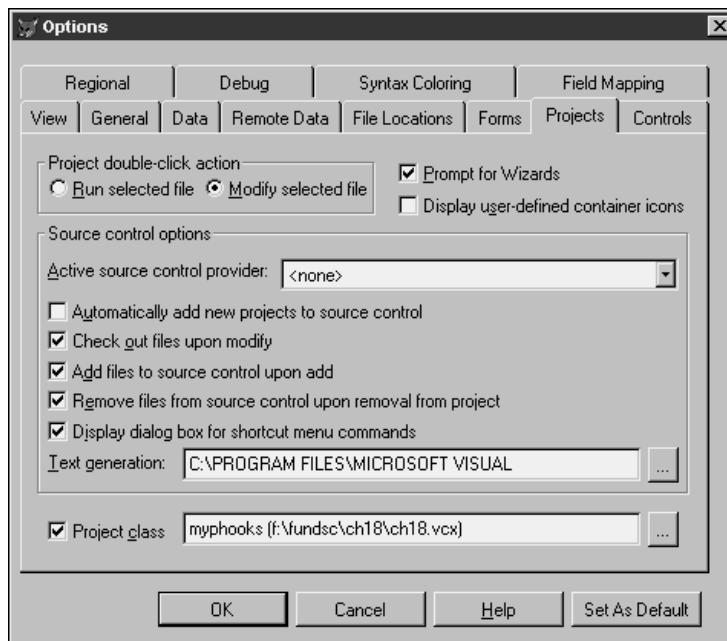
When you create a new project, the global ProjectHook class is automatically added to the project. Of course, you can go into the Project Info dialog and remove it, but that's awkward. A better way is to simply issue the command:

```
create project MYPROJ noprojecthook
```

On the flip side, you might want to work with a project but not display the Project Manager window while doing so. This would be particularly applicable if you were building tools for other developers, and automatically creating new projects or editing existing projects as part of the process. You can use the following commands to perform these actions without any visible display:

```
create project MYPROJ noshow  
modify project MYPROJ noshow
```

The key benefit is that the project and project hook objects are still instantiated, so your program will still have access to all the functionality of your ProjectHook class.



**Figure 18.6.** Setting a ProjectHook class globally.

## Project hook examples

There are two examples of project hooks with the Visual FoxPro samples. The Activity\_Tracker sample, in the Program Files\Microsoft Visual Studio\MSDN98\98Vsa\1033\Samples\VFP98\Solution\Tahoe directory, tracks actions done with a project and saves relevant information to a log file. It's a good, easy-to-follow example.

The Visual FoxPro Application Wizard comes with its own source code, and you can open it to discover how project hooks are used to update a project file with information gathered from the Wizard's interface.

My technical editor has done a lot with project hooks, and has suggested the following ideas for using them:

- Create a developer toolbar for common project operations. You could set up different buttons for "one-click operations," such as rebuilding all files, building an .EXE, and other frequent processes.
- Incorporate changes to various project options and development settings before building. For example, you might want to turn on debug info and asserts for a test build, but turn them off and increment version numbers for production builds.
- Initialize project-specific items. For example, you might want to assign one set of Field Mapping classes for one project, and another set for a different project. These could be set automatically when the project is opened.
- A variety of project utilities, including creating documentation, searching tools throughout a project's components, file packing (.SCX, .VCX, and .FRX), and ZIPping for backup and distribution.

# Chapter 19

## Using the Debugger

FoxPro developers through version 3.0 were saddled with a fairly limited set of debugging tools: a two-pane window that displayed the values of variables, and a code tracing window. When Fox developers looked over the shoulder of a VB or Visual C developer, the usual feeling was one of envy—for a robust, full-featured debugger. One of the major improvements in version 5.0 was a tool similar to other debuggers. Pound for pound, the Visual FoxPro 6.0 Debugger has probably as many features and great capabilities as any component of Visual FoxPro. In this chapter, I'll take you through the entire tool. If you've not used a debugger seriously, fasten your seatbelts.

Debugging is one of those dirty secrets that all developers keep hidden from their users and customers. Furthermore, many developers don't want to admit how much they have to debug, and, thus, correlate a high level of debugging skill with a low level of programming skill. As a result, debugging techniques don't get as much airplay. Well, that's all nonsense.

I remember a story about a class of fifth graders that is taken to the racetrack for a day. At the end of the day, after watching practice laps, watching the pit crews, meeting the drivers and others involved in running the facility, the teacher asked who was the best driver they met. Various students offered their opinions, but the teacher corrected them all. "The bus driver who took us to the track and then returned us to school at the end of the day. He's a true professional—he drove safely and carefully, and got us where we wanted to go without any problems. While a driver who runs into problems and escapes by the skin of his teeth may be flashy, the driver who never gets into trouble in the first place is better."

Similarly, the programmer who knows that bugs are inevitable, and who can masterfully debug his application, is much more worthy of your praise than the guy who claims he hardly ever needs to use it.

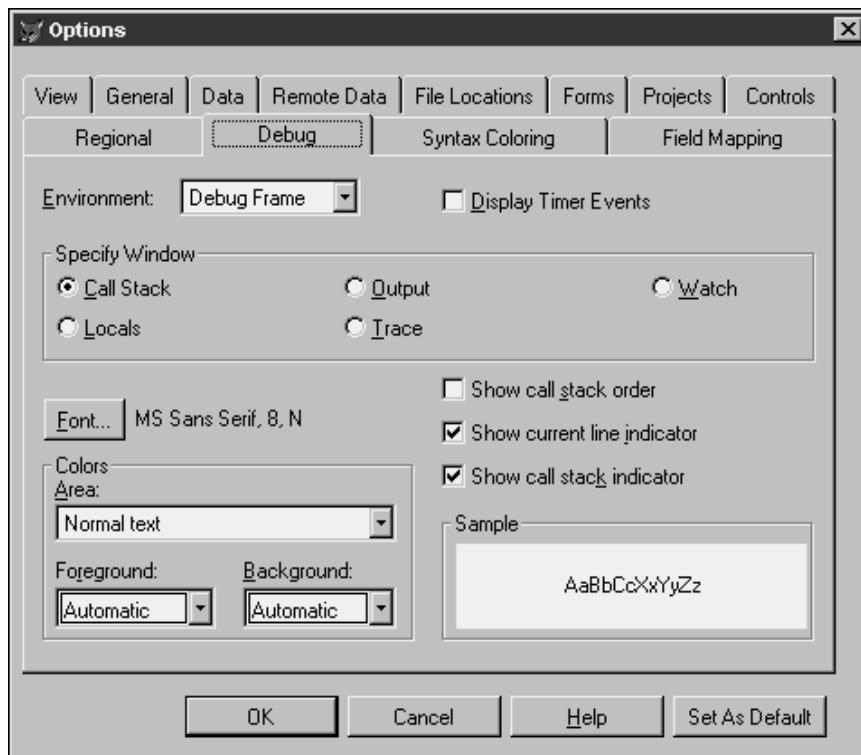
In this chapter, I'm going to show you how to configure the Debugger to suit your style, and then walk through each of its components. While I'm doing so, I'll discuss some techniques—some that I use and others that my tech editor, Doug, relies on—that you can use in your day-to-day work. Finally, I'll wrap up with a lecture on the debugging process in general. Here, more than anywhere else, having the right mindset is critical to being a successful debugger.

You'll want to read this chapter at least twice because it's circular in nature. Debugging is kind of a Catch-22 situation—simply describing each component without putting its use in context is somewhat useless, so I provided a brief vignette as well. However, in doing so, I needed to refer to another component that I haven't discussed yet.

### Configuring the Debugger

The Debugger consists of five windows: Call Stack, Output, Watch, Locals, and Trace. How these windows work—with each other as well as with Fox—is determined by configuration

settings. The Options dialog (found by selecting Tools, Options) has an entire tab (Debug) devoted just to the Debugger configuration settings. See **Figure 19.1**.

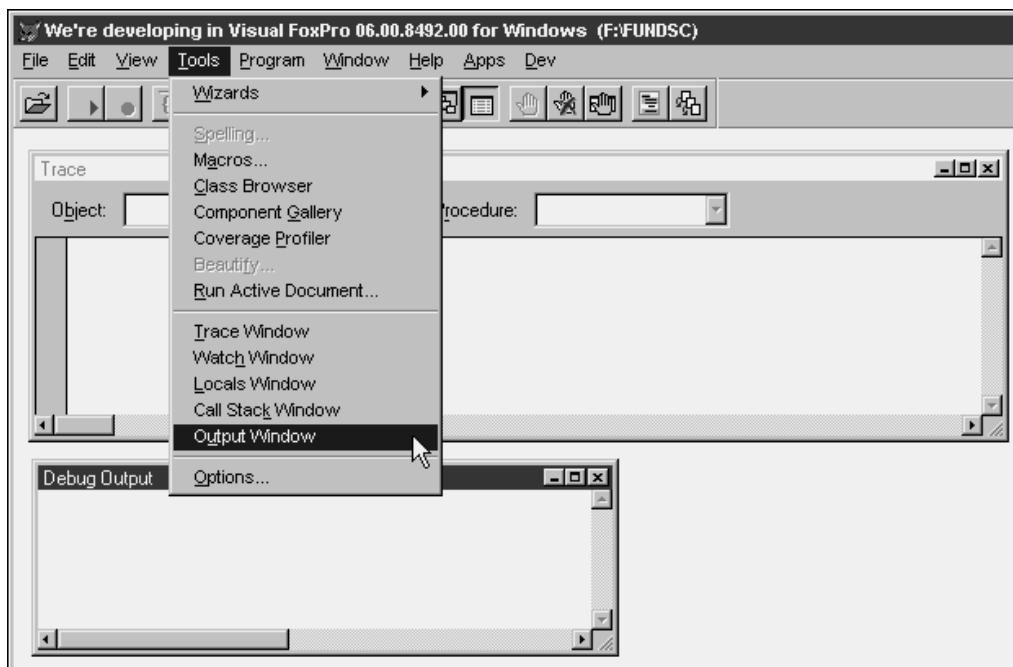


**Figure 19.1.** The Options dialog has an entire tab dedicated to the Debugger.

## Choosing a debugging frame

The first thing you're going to want to do is to choose how the Debugger window is opened. You can open the Debugger in two modes—one where the windows are all part of the Visual FoxPro desktop, just like the Command window and Data Session windows, or in a separate application that displays as a separate window as well as in a separate item in the task bar.

You choose which mode by selecting an item in the Environment combo box. If you choose the FoxPro Frame, debug windows will appear in the FoxPro desktop, and the Tools menu will have five menu options—one for each window. See **Figure 19.2**. Selecting the Debug Frame means there will be only one menu option on the Tools menu. Selecting this menu option will open a window separate from the VFP desktop in which debug windows will appear, as shown in **Figure 19.3**.



**Figure 19.2.** Choosing the FoxPro Frame environment in the Options dialog allows debug windows to be opened in the VFP desktop.

Note that if you have any debug windows open, or if the Debugger toolbar is open, you won't be able to change the value in the environment combo box.

## Configuring windows

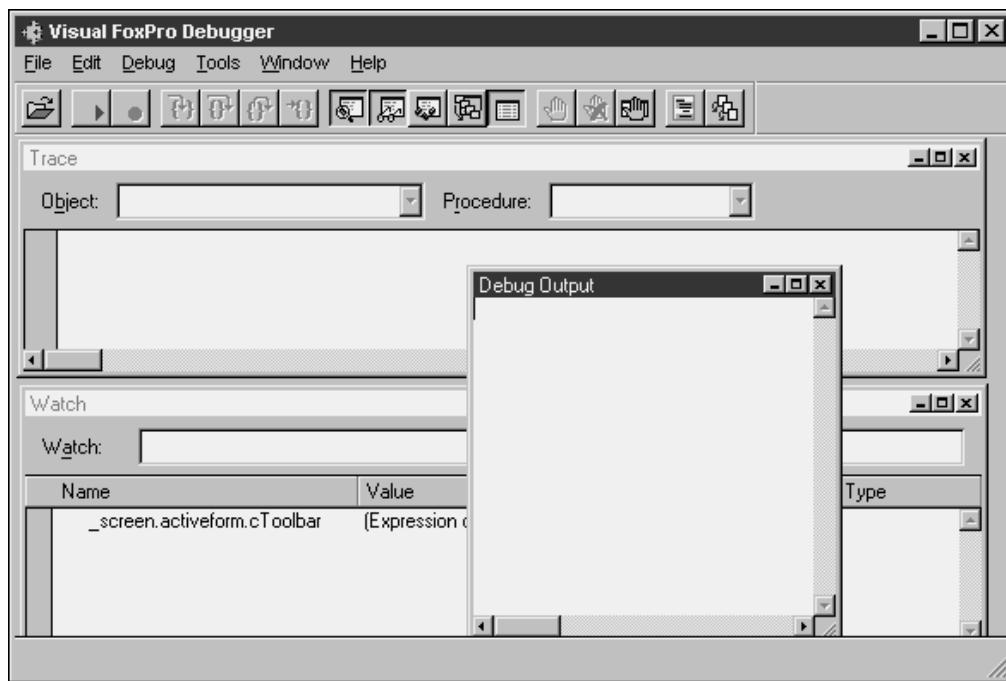
The interface in the bottom two-thirds of the Options dialog's Debug tab is a little tricky. First, you select one of the five windows through the Specify Window option group. Once you do so, the values of all controls underneath the option group relate to the selected window. The actual controls that display under the option group vary according to which window was selected.

### Common settings

You can set the font and colors properties for each window. Clicking the Font button will open the Font dialog that you see everywhere in Windows—you can select a font, its style, and size.

You can also choose the color properties for unselected (“Normal”) and selected text in the window in question. To do so, select the type of text—Normal or Selected—in the Areas combo box, and then set the Foreground and Background colors through those combo boxes.

The choices you make appear in the Sample read-only text box (or is it a fancy label?).



**Figure 19.3.** Choosing the Debug Frame environment in the Options dialog allows debug windows to be opened in a separate window.

### Settings for Call Stack window

Three check boxes appear in the right side of the Debug tab when you select the Call Stack option button (see Figure 19.1):

- Show call stack order: Checking this check box will display a number next to each program listed in the Call Stack window. The number 1 indicates the first program in the call stack; each subsequent number indicates the next program called; and the highest number indicates the currently executing program.
- Show current line indicator: Checking this check box will display a right-pointing arrow in the Call Stack window next to the program that contains the currently executing line.
- Show call stack indicator: Checking this check box will display a right-pointing arrow in the Call Stack window next to the program that is displayed in the Trace window. Well, that's what the documentation says. If I uncheck the current line indicator and then check the call stack indicator, I get an arrow in the Trace window but nothing in the Call Stack window. Either these controls are broken, or the doc is.

### Settings for Output window

A check box, a file selection text box and ellipsis button, and an option group appear in the right side of the Debug tab when you select the Output option button. See **Figure 19.4**.

Checking the Log Debug Output check box will enable the other controls. You can also check the Log Debug Output check box to send data that is sent to the output to a file. You can enter the name of a file in the text box or select an existing file by clicking the ellipsis button and using the Open dialog. You can choose to overwrite any existing information in that file or simply append the new data to the existing contents of the file.

This log file is a global log file—data from any application that is sent to the output window will be sent to this same log file. While it's handy to set this option once, you can also generate huge files if you're not paying attention. Choose Overwrite or Append carefully! You can also manually send Debug Output to a file by using the SET DEBUGOUT TO <file> command in your code.

### Settings for Watch window

There are no controls specific to the Watch window.



**Figure 19.4.** The Debug tab with the Output controls.

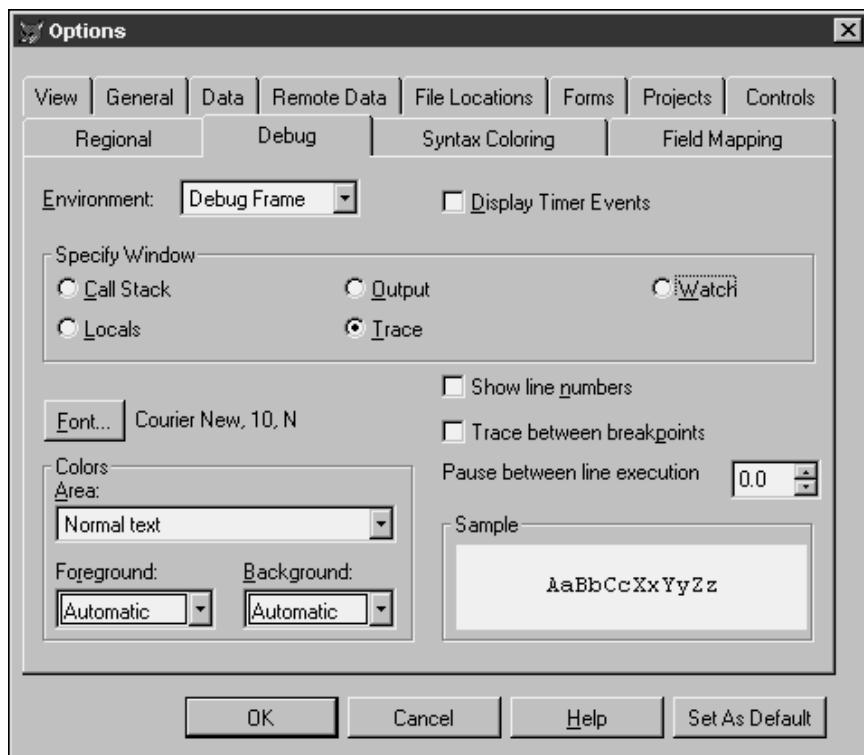
### Settings for Locals window

There are no controls specific to the Locals window.

### Settings for Trace window

Two check boxes and a spinner appear in the right side of the Debug tab when you select the Trace option button. See **Figure 19.5**.

- Show line numbers: Displays line numbers in the left margin of the code in the Trace window.
- Trace between breakpoints: Executes code between breakpoints at throttle speed (see next item). For example, if you set the throttle to 1 second, you could watch your code execute (the executing line would be highlighted) in the Trace window by observing the highlight moving from line to line, one second per line.
- Pause between line execution: Specifies the delay in seconds between the execution of each line of code—in other words, the throttle.



**Figure 19.5.** The Debug tab with the Trace window controls.

## Other options

Checking the Display Timer Events check box will display, in the Trace window, timer events that occur while your program is running. For example, suppose you have a timer on a form and it is set to fire every five seconds. When it fires, a refresh command is executed. When you are running the form, you'll see the various code statements in the Trace window, and every five seconds you'll see a refresh statement as well—if you've checked this check box, that is. Unless you've got a very specific reason for having this on, you'll want to keep it off.

## The five Debugger windows

The Visual FoxPro 6.0 Debugger consists of five independent windows, as well as a toolbar that allows you to get quick access to a variety of debugging functions. The Debug window that appears when you run the debugger in a Debug Frame also has a menu that gives you quick access to Debugger functions.

### Trace window

The Trace window allows you to display the lines of code being executed, and to optionally set breakpoints at specific lines of code in your program. (I'll discuss breakpoints in a bit.) This can come in handy when your program crashes, or when you want to intentionally stop execution to look at a line or segment of code.

To use the Trace window, follow these steps as an example:

1. Open the Trace window (in either frame).
2. Create a program with a WAIT WINDOW that displays a bad expression, like so:

```
wait window m.lcNoSuchVariable
```

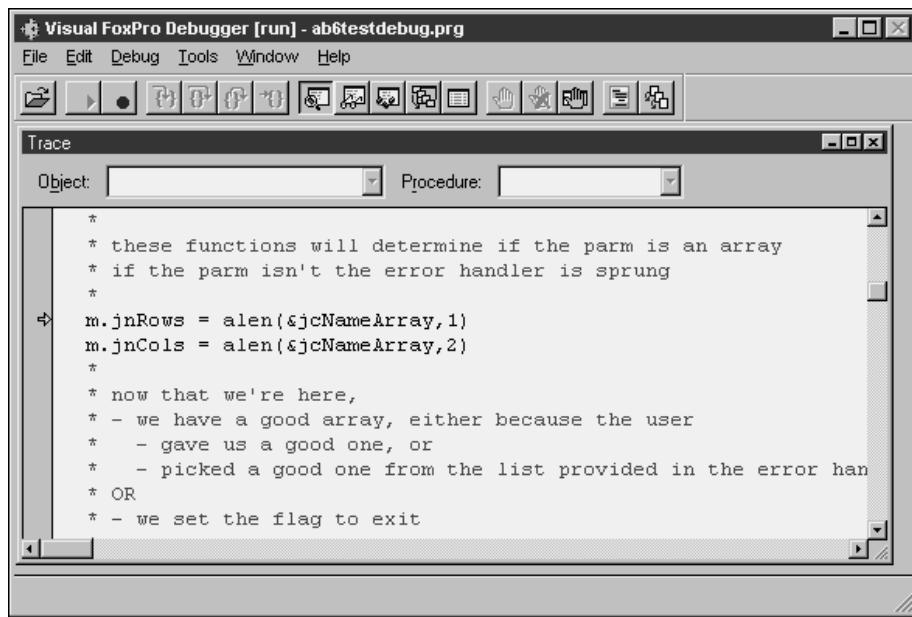
3. Open the program in the Trace window, using the Open button in the debugging toolbar.

4. Run the program and watch your program generate an error message with Cancel, Suspend, Ignore, and Help buttons:

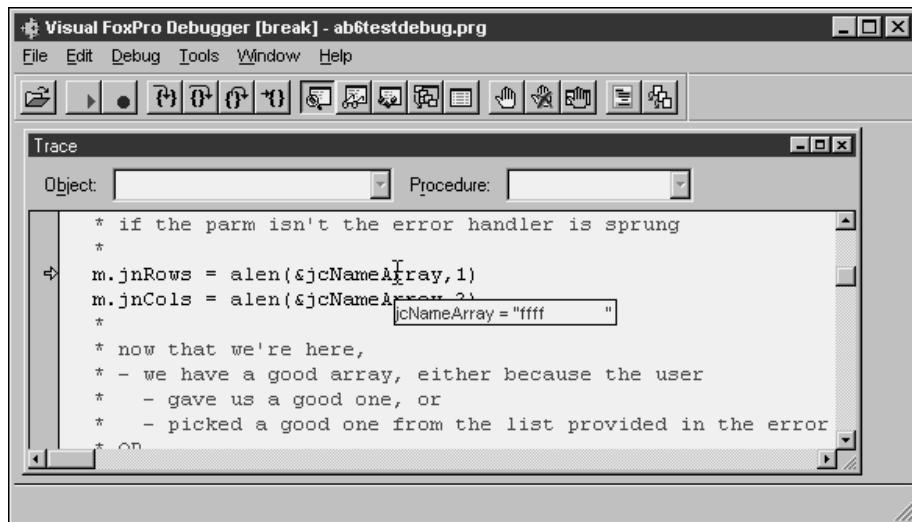
```
Variable m.lcNoSuchVariable is not found
```

5. Click the Suspend button.
6. Switch over to the Trace window. You'll see the WAIT WINDOW line of code highlighted with a yellow arrow in the left margin.

This particular error message is easy to see and resolve—perhaps a typo or some such problem. If, on the other hand, the variable exists but caused another problem, such as a “data type mismatch,” you can hover your mouse cursor over the variable name in the Trace window and see what the actual value is, as shown in **Figures 19.6** and **19.7**. In Figure 19.6, the variable “jcNameArray” exists, but it is a character expression, thus causing an error. In Figure 19.7, the mouse screen tip shows that the variable is indeed a character string.



**Figure 19.6.** The yellow arrow signifies the current line of code in the Trace window.



**Figure 19.7.** Once code is suspended in the Trace window, you can hover your cursor over an expression and the evaluated value will display in a screen tip. Here, the screen tip shows that the array name "ffff" was passed to the AB6TESTDEBUG program.

---

When you’re done, you can continue running your program by clicking the Resume button in the debug toolbar or selecting Resume from the shortcut menu (described later).

### Throttle

If you have the Trace window open and a complex routine is executing, you might be able to see the lines of code scrolling by with the yellow arrow blipping down the left margin of the window. You might be thinking that it would be handy to see the code go by a bit more slowly. You can do this by setting the throttle of the Trace window.

Select the Debug, Throttle command from the Debugger menu and enter a value, in seconds, that you want Visual FoxPro to wait in between executing commands. This is the same value that was called “Pause between line execution” in the Debug tab of the Tools, Options dialog. Usually a quarter-second to a half-second is plenty, although your mileage may vary depending on your personality, preferences, and the horsepower of your machine.

Then, when you run your routine with the Trace window open, the code will scroll by slowly, according to the time interval you specified.

Of course, this brings up another question, doesn’t it? Once you see code go by, how do you get your program to stop at an interesting point? You could just press Escape, but that could be dangerous to the environment—what you’d like to do is temporarily suspend execution instead.

Many developers assign the SUSPEND command to a hotkey, like so:

```
ON KEY LABEL F12 SUSPEND
```

Then, as your program is executing and code is scrolling by in the Trace window, you can just press F12.

You might even embed this command in your program, instead of having to type it in the Command window. If you took this route, you’d probably want to bracket the code so it was activated only when the program was in development mode or when you had logged on.

Again, once you’re done with whatever you wanted to do with your suspended program, you can continue execution by selecting Resume.

### Stepping through code

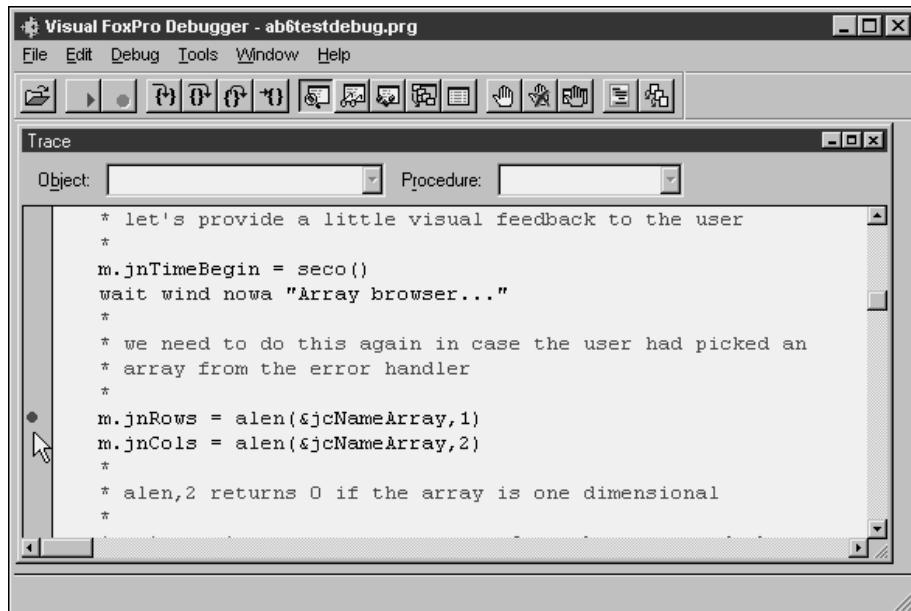
So now that you’ve got the Trace window meandering through your code according to the Throttle value, and you can suspend your program at will, you might want to examine a particular routine and its values, one line at a time. Here’s how.

First, set up your Trace window, Throttle, and Suspend hotkey as necessary. Run your program until it gets to the point of interest—say, a looping construct that is going bonkers on you. Suspend your program and use your mouse to hover over the values of interest. When you want to move to the next line, click the Step Into button on the Debug toolbar.

### Breakpoints

There might be times when you want to stop the execution of a program without an error causing it. You can set a breakpoint by double-clicking in the left margin of the Trace window on a line of code. When you do, you get a red dot in the margin, as shown in **Figure 19.8**.

Now, run your program (without any code that will cause it to generate an error). You'll see the program stop without warning, which is your cue to switch over to the Trace window. You'll see the yellow arrow in the left margin on the same line of code on which you set your breakpoint. You can now examine values or perform other operations, as you would have earlier. The difference is that you can control where the program breaks.



**Figure 19.8.** The red dot signifies a breakpoint in the Trace window.

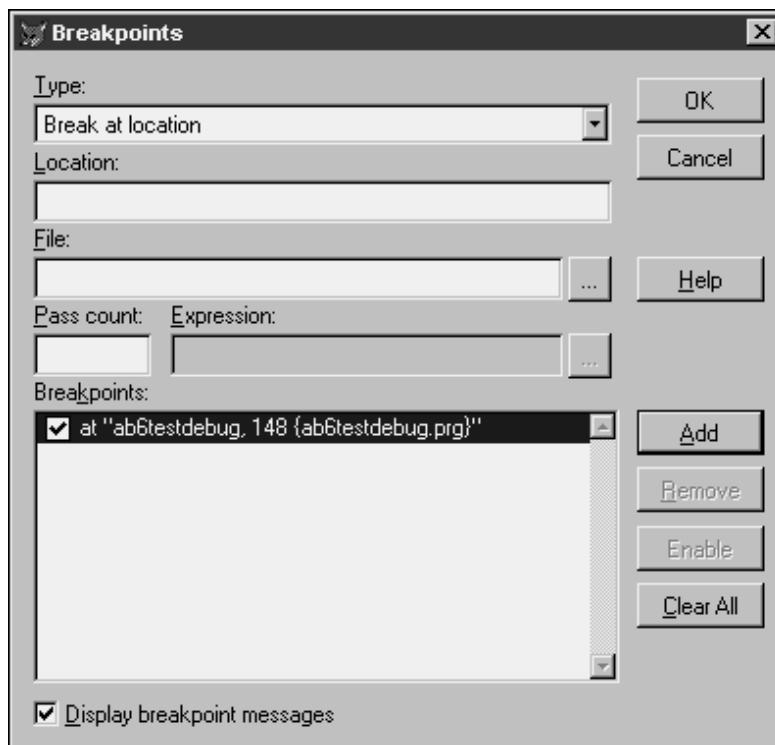
### Breakpoints dialog

While setting a breakpoint or two in the Trace window is nice for occasionally debugging, you'll often want to set up a group of breakpoints in a long or complex routine. Instead of having to create one breakpoint after another while testing, and then repeat it the next time you test, you can set up a group of breakpoints and store that group for later recall.

Even better—you're not limited to setting breakpoints at specific lines of code. You can also set breakpoints when an expression becomes true, or at a location only if an expression is true when that line of code is executed, or when an expression changes. And, of course, you can create a group of breakpoints that mix and match these different types of breakpoints. This is all done in the Breakpoints dialog, which is shown in **Figure 19.9**.

You can open the Breakpoints dialog by selecting the Tools, Breakpoints menu option *in the Debug Frame menu*, or by clicking the Breakpoints dialog button in the Debugger toolbar.

To create a breakpoint in the Breakpoints dialog, first select the Type. The steps you take next depend on which type of breakpoint you've chosen.



**Figure 19.9.** The Breakpoints dialog.

### **Break at location**

If you select Break at location, the Location, File, and Pass Count text boxes become enabled. You can enter the name of a program, a procedure, or a specific method or event. You can optionally add a line number in the routine where you want the breakpoint to be set after the name of the routine, like so:

**ab6testdebug, 148**

If you just enter the name of the routine, the breakpoint will be set at the beginning of the routine. Enter the file name (or select a file using the ellipsis button) in the File text box.

If you are setting a breakpoint in a looping construct, you can set the breakpoint to fire only after a certain number of iterations by entering a value in the Pass count text box. For example, if you want your program to be suspended at the 13th time it's executed, enter "13" into the Pass count text box.

***Break if expression is true***

If you select Break if expression is true, the Expression text box and ellipsis button become enabled. You can enter any valid Visual FoxPro expression in the text box, or create an expression with the Expression Builder that is opened if you click the ellipsis button.

What types of expressions might you want to break on?

The first type of expression most people think about is comparing a variable with a constant, such as when an amount is equal to zero, or, alternatively, when the amount becomes non-zero. You can compare two variables with each other, such as a gross amount and a net amount. And you can concatenate two expressions—when a date reaches a certain value and an amount is greater than a certain amount.

However, you're not limited to just these types of expressions. How about checking to see if you're at beginning or end of file in a table, or if you've got the correct cursor in the current work area, or have an index set? You could check to make sure you have buffering set the way you expected it to be, or if you have Deleted set to ON or OFF, or what the setting of SET EXACT is.

Another favorite is to look for a specific text string in the name of the currently executed program, and break there—that way, you can run your application until you get to the area of interest, and then break. If you specify an expression involving a *local* variable instead of a hard-coded value, the breakpoint will also fire when the execution dives into another routine—in which case the variable won't be in scope any longer.

***Break at location if expression is true***

If you select Break at location if expression is true, the Location, File, and Pass Count text boxes become enabled as well as the Expression text box and Expression ellipsis button. This works like creating a compound expression on Break at location *and* Break if expression is true, so all of the same ideas work here in combination.

***Break when expression changes***

If you select Break when expression changes, the Expression text box and Expression ellipsis buttons become enabled, just as with Break if expression is true. However, with this break type, you don't have to know exactly the value of the expression that you're matching. Instead, Visual FoxPro will watch for any change in the expression. Like "Break if expression is true," you can either enter your own expression in the text box or create an expression using the Expression Builder.

This type of breakpoint is handy when you're chasing down issues where a value or a setting changes when you don't want it to change. You don't particularly care what it's changed to, you just want to find out where it's being changed. For example, suppose a command you're using to match a value is all of a sudden not working—and you determine that the setting of EXACT has been changed to ON. You might use the expression

```
SET ("EXACT")
```

to determine when the setting of EXACT is being changed. Similarly, suppose you are moving the record pointer in a cursor, table, or view, and suddenly you get mysterious error messages

indicating that you don't have a table open, or that you're not in the place you thought you were. You can use the expression

**ALIAS ()**

to determine where you're switching work areas.

### **Managing breakpoints**

Once you've entered all the values needed for the breakpoint you're creating, click the Add button and the breakpoint will appear in the Breakpoints list box at the bottom of the Breakpoints dialog. If you made a mistake, you can delete the breakpoint from the Breakpoints list box by selecting the row in the list box and then clicking the Remove button. Note that you will not be prompted to confirm your choice. Further, you can delete all breakpoints from the list box by clicking Clear All. Like Remove, you will not be prompted to confirm your choice after clicking Clear All—it will just blast them away.

Once you've created a breakpoint, you'll want to enable it. It's enabled by default (you can tell by the check mark in the check box in the breakpoint's item in the list box), but if you've disabled it, you can enable it by clicking the check box to the left of the breakpoint in the list box, or highlighting the list box and clicking the Enable button.

If you want to keep a breakpoint in a group, but temporarily disable it, click the check box to the left of the breakpoint in the list box, or highlight the list box and click the Disable button.

To store this group of breakpoints, use the File, Save Configuration menu option.

When you set a breakpoint (other than a "Break at location" breakpoint), you'll get a breakpoint message when the breakpoint hits. For example, in **Figure 19.10**, I set a breakpoint to fire when the expression "m.lcY" changes. This memory variable is simply assigned a value in the Click() method of a command button.

When I click the command button, the variable m.lcY is assigned a value, the breakpoint fires, and the message box shown in **Figure 19.11** is displayed.

If you want to prevent these breakpoint message boxes from displaying, uncheck the Display breakpoint messages check box.

This affects *all* breakpoints, not just the breakpoint that's highlighted in the list box.

### **Trace window shortcut menu**

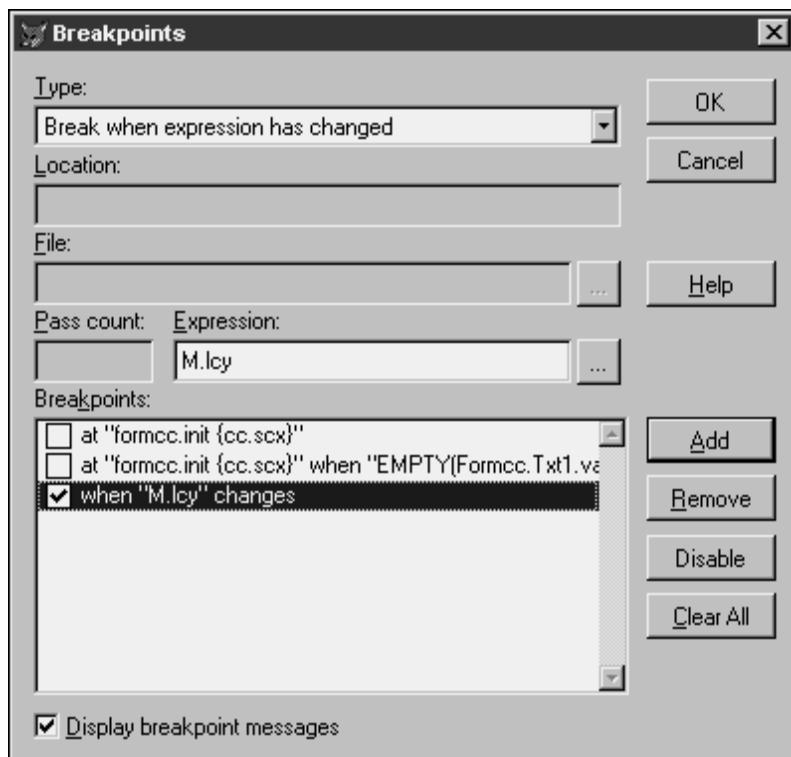
If you right-click in the Trace window, you'll get a context menu that contains a number of helpful options.

#### **Open**

Displays the Open dialog box so you can open a file for viewing in the Trace window.

#### **Resume**

Oftentimes, you'll want to simply examine part of a program and then continue with it. You'll temporarily stop execution by setting a breakpoint (or selecting Suspend in the Cancel, Suspend, Ignore error dialog). To start up again, you can just select Resume here.



*Figure 19.10.* Setting a breakpoint when the value of the memory variable *m.lcy* changes.



*Figure 19.11.* A message box displays when a breakpoint is reached.

### **Step Into**

This command will execute the next line of code if you are stepping through a routine line by line.

### **Step Over**

This command will execute the next line of code, just as Step Into, but with one difference. If the line of code to be executed calls another routine, that routine will be called in the background and then control continues with the next line.

### **Step Out**

This command will continue executing code in a procedure without stepping through code line by line. Suppose you’re stepping through a routine, ProgA, and that routine calls a subroutine, ProgB. You’ve started stepping through the lines of code in ProgB, and then realize that you don’t need to—you’d like to finish up ProgB and then resume stepping through, line by line, the code in ProgA again. Step Out will execute the rest of the lines of code in ProgB without pausing, and then suspend program execution once ProgA is reached.

### **Set Next Statement**

Suppose you’ve suspended execution and examined part of a routine. Now you need to skip one or more lines of code from where execution was suspended—perhaps you want to bypass a line or segment that you know is going to fail, or perhaps you have manually set some values and you don’t want the next few lines of code to alter those values.

But you don’t have to skip forward! You can also use this to go back and re-execute some code after you’ve changed something, such as initializing a variable *correctly!* (Good luck remembering to change it correctly in your code later! I usually repeat this mistake twice before remembering...) Or perhaps you just want to replay a sequence of code in super-slow-mo because it went by on the head-cams too quickly the first time around.

You can place your cursor on another line of code in the Trace window, and then execute this menu option in order to resume execution where the cursor is, not where the routine was suspended.

### **Run to Cursor**

No, this isn’t a fancy way of saying that you want to store a bunch of values to a cursor while the Trace window is open. Instead, suppose that you have suspended execution, and now you want to run a segment of code—perhaps just the next few lines of the routine. However, when that batch of lines is finished executing, you’d like the program to be suspended again.

You can place your cursor on the last line of the section of code you want to execute, and then select this menu option, and just that section of code will be executed. In other words, you’re going to continue to run the program until you reach the line where you had just placed the cursor—thus, “run to cursor.” Perhaps “run to cursor location” or “execute to marked line” would have been a bit clearer, but once you try it a few times, you won’t think twice about it.

### **Trace Between Breaks**

Suppose you have several breakpoints set in a routine, and you want to step through the code between breakpoints A and B, and then between C and D—but you don’t want to laboriously

step through line-by-line between B and C. You can toggle tracing with this menu option. First, you'd execute your code until you hit the first breakpoint. Then you'd toggle this on, so you can watch each subsequent line execute. Once you reached breakpoint B, you'd toggle this off, and then continue execution until reaching breakpoint C, where you'd toggle it back on.

### **Docking View**

If this menu command is checked, you can dock the window inside the debugger frame. If this menu option is not checked, you won't be able to dock the window.

### **Hide**

Simply hides the Trace window.

### **Font**

Opens the Font dialog box so you can change the font name, style, and size.

## **Watch window**

The Watch window allows you to enter the name of a variable, and then see what value that expression evaluates to.

### **Creating a watch expression**

To set a watch expression, follow these steps:

1. Open the Watch window.
2. Enter the name of a variable or other expression in the text box at the top of the window.
3. Press Enter.

The expression you entered will be moved to the list box underneath the text box, and the value will initially be set to "Expression cannot be evaluated" (unless you currently have a variable in memory that matches that expression).

You can also drag expressions from the Trace window to the Watch window if you've got both windows open. Pretty awesome, eh?

### **Manipulating watch expressions**

One hidden mechanism in the Watch window is the ability to change the value of an expression. You can select a watch expression in the Watch window, and then either change the expression or the value of the expression. To do so, double-click the expression or the value so that it's put into edit mode, type a new expression or value, and then press Enter.

### **Using watch expressions**

So now you know how to create and change watch expressions. How do you use them in the living hell that we all know as "debugging at 2:30 in the morning"?

First, you can use the Watch window just to determine the value of expressions as lines of code are being executed. If you set a breakpoint on a line of code, execution will pause so that

you can either look at existing watch expressions or enter new ones. Sure, you can simply hover your cursor over expressions that are explicitly listed in the code, but you can't do that if you want to look at other expressions at the same time.

Second, if you play your cards right, you can actively change values while running your routine. This can be handy if you've identified an error and want to temporarily fix it instead of having to stop execution, fix your program, and then run it again. For example, suppose you have a variable that, instead of being initialized to a numeric zero, is erroneously initialized to an empty character string. Later you want to add it to another number, but the addition crashes because you can't add a character string to a number. You can suspend execution after the variable is assigned the character string, change it to a zero, and then continue execution of your routine—the addition statement will succeed this time.

You can set or clear a breakpoint on a Watch expression—one of those “when expression changes” types of breakpoints—by double-clicking in the gray band to the left of the expression. You can also set a “By Location” breakpoint by using an expression such as “`FunctionName`” or “`MethodName`” \$ `PROGRAM()` and setting a breakpoint on the expression. For example:

```
"DoSecondIncrement" $ PROGRAM()
```

### **Watch window shortcut menu**

As with the Trace window, the Watch window has a context menu, albeit a shorter one.

#### ***Insert Watch***

This is a little tricky—executing this menu option inserts a new blank watch expression. You might think this would add a new row to the list box, but it actually just sets focus to the text box. Go figure, ya know?

#### ***Delete Watch***

You can delete an existing Watch expression by highlighting the expression in the list box and then selecting this menu option, or by selecting the expression and pressing the Delete key. You won't be asked to confirm your action. Note that you have to delete watch expressions one at a time because the list box doesn't allow you to select multiple expressions.

#### ***Docking View***

If this menu command is checked, you can dock the window inside the debugger frame. If this menu option is not checked, you won't be able to dock the window.

#### ***Hide***

Simply hides the Watch window.

#### ***Font***

Opens the Font dialog box so you can change the font name, style, and size.

## Locals window

The Locals window is sort of like the Watch window on steroids and amphetamines. Instead of just displaying the expressions that you want to see, the Locals window displays a list of every available variable in the current program stack. This includes a complete list of the object model in memory—for example, if you are running a form, you can drill down into the form to a command button that's on a particular page in a page frame and see the current value of each of its several dozen properties. This capability even works for arrays and collections—you'll see a plus sign next to the name of a variable, indicating that you can drill down to a lower level, whether elements of an array or items in a collection.

## Call Stack window

A *call stack* is the list of programs that were executed in order to get to a certain point. If PROG1 called PROG2, which in turn called PROG3, the list of those three programs is the call stack.

To demonstrate the use of the Call Stack window, I created three programs: AA, BB, and CC. Program AA assigns values to three local variables, and then calls CC (not BB). Program CC assigns values to local variables with the same names as in AA, and then calls BB. Program BB again assigns values to local variables with the same names as in AA and CC, and I deliberately add a character string and a numeric value to generate an error in the third program in line:

```
* aa.prg
local m.lnX, m.lnY, m.lnZ
m.lnX = 1
m.lnY = 2
m.lnZ = 3
do CC.PRG

* cc.prg
local m.lcX, m.lcY, m.lcZ
m.lcX = "10"
m.lcY = "20"
m.lcZ = "30"
do BB.PRG

* bb.prg
local m.lcX, m.lcY, m.lcZ
m.lcX = "AAA"
m.lcY = "BBB" + 123
m.lcZ = "CCC"
```

I ran program AA and generated the usual Cancel, Suspend, Ignore error dialog. I suspended, and then opened the Call Stack window in the Debugger, as shown in **Figure 19.12**. I also opened the Trace window to ensure that the specific line that caused the error was the one I intended.

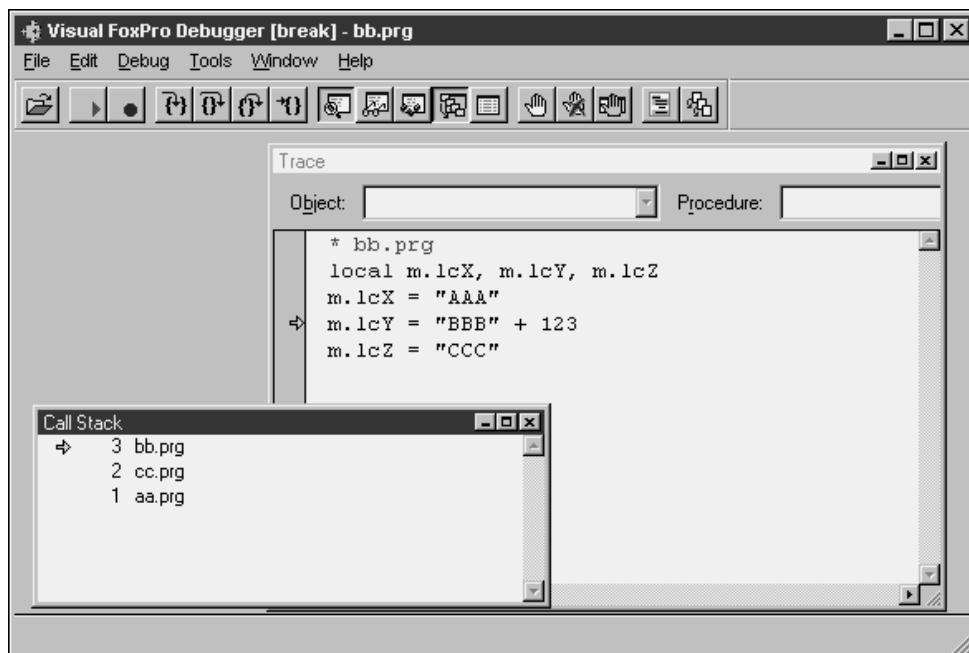
The programs called are all listed in the Call Stack window. Checking the Show call stack indicator check box in the Tools, Options dialog caused the numbers to be displayed next to the names of the programs. As you can see, AA was called first, then CC, then BB. The arrow next

to program BB is displayed because I checked the Show current line indicator check box in Tools, Options.

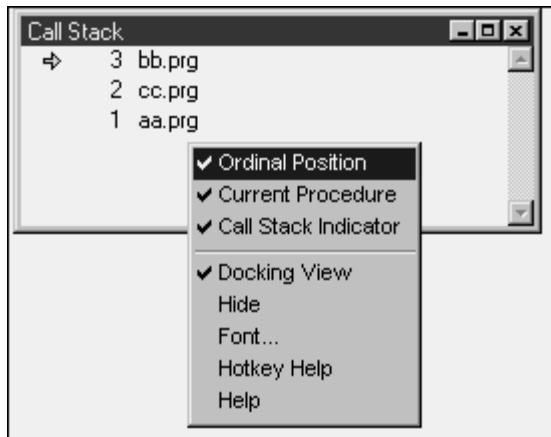
### Call Stack window shortcut menu

Right-clicking in the Call Stack window, as shown in **Figure 19.13**, produces a menu with several useful options. Each of the first three map to the check boxes in the Debug tab of the Tools, Options dialog. Clicking the Ordinal Position menu option will turn the numbers next to the programs listed in the Call Stack window on or off. Note that the documentation for the Call Stack window in the Debug tab never refers to these values as “Ordinal Positions.” Clicking the Current Procedure menu option will turn the yellow arrow pointing to the currently executing procedure on or off. And clicking the Call Stack Indicator menu option will turn the yellow arrow in the Call Stack window on or off to indicate the procedure displayed in the Trace window. If the current line and the call stack procedure are the same, you’ll see only the current line indicator, so messing with this option won’t appear to do anything.

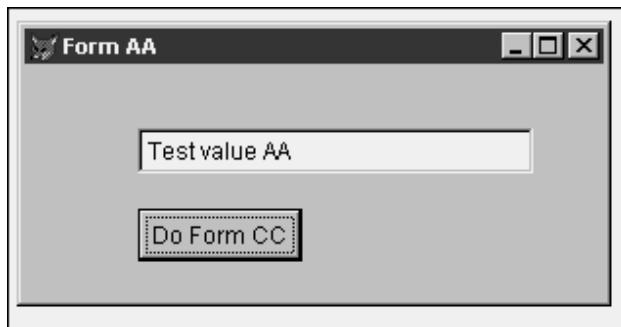
So that’s how it works with a set of programs. What about when forms and events and methods get involved? Next, I created a series of three forms, innovatively called FormAA, FormBB, and FormCC. Each has a text box, named txt1, and a command button, command1. See **Figure 19.14**.



**Figure 19.12.** The Call Stack window shows you a list of programs that have been executed to get to the current state.



**Figure 19.13.** Right-clicking in the Call Stack window displays the Call Stack context menu.



**Figure 19.14.** The sample form used in examining how the Call Stack window works with forms.

In each of the form's Init() methods, I set the value of a text box, txt1. Then I have FormAA's command button call FormCC, and FormCC's command button call FormBB. In the Init() of FormBB, I reference a text box, txt2, that didn't exist, which causes an error. The results of the Call Stack and Trace windows are shown in **Figure 19.15**.

## Debug Output window

The Debug Output window is my personal favorite. The Watch window allows you to look at the values of expressions while your code is running, and that's all fine and good—but once your code has run, the Watch window will only hold the values that those expressions last held.

The Debug Output window, on the other hand, allows you to output the value of expressions (or even just text strings) while your program is running, and it keeps those values

available after the routine is finished. You can even capture the values to a text file if you want. This provides not just a snapshot of your application at a specific time, but also a history over time.

To output (or echo, if you prefer) the value of an expression to the Debug Output window, use this command in your code:

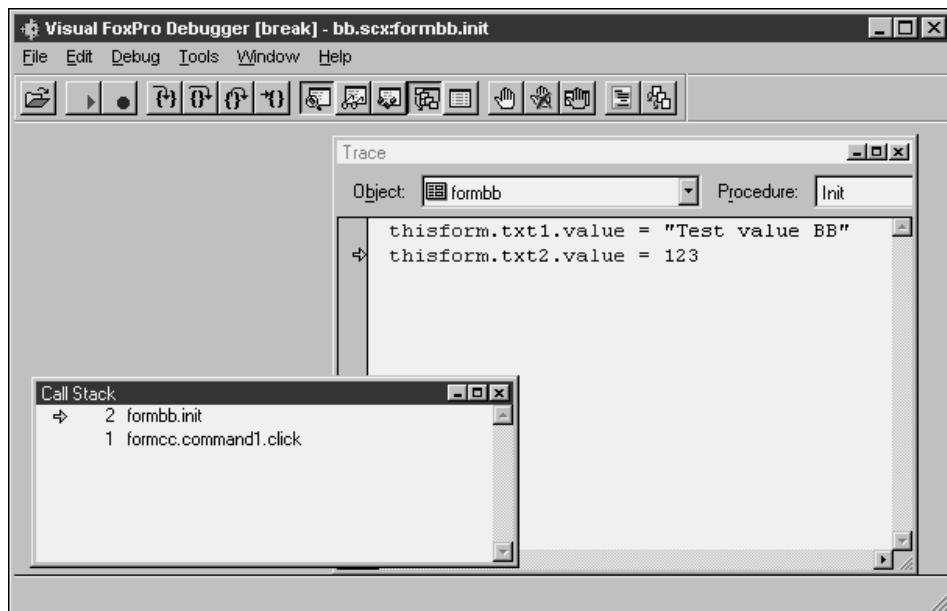
```
DEBUGOUT <expression>
```

For example:

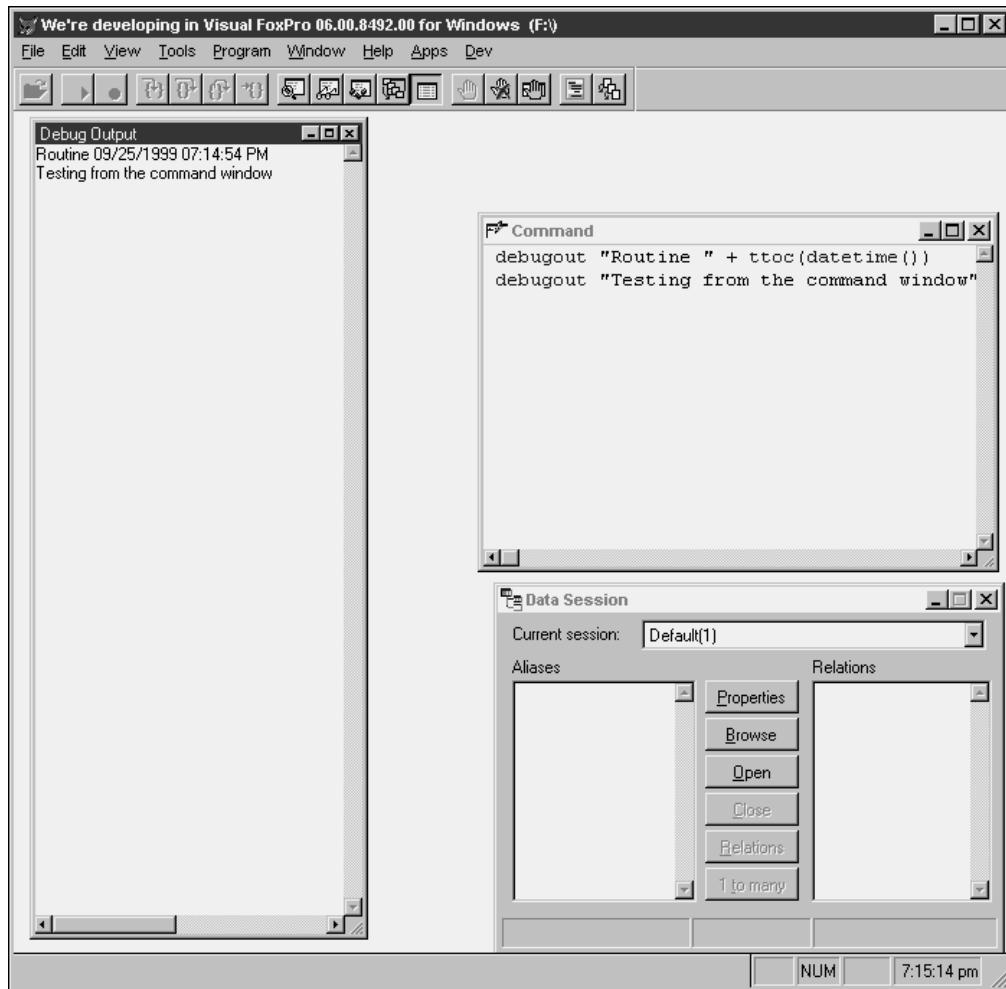
```
DEBUGOUT m.lnX
DEBUGOUT alias()
DEBUGOUT cursortgetprop("Buffering")
DEBUGOUT "Routine " + PROG() + " " + TTOC(datetime())
```

These are all valid expressions you could echo to the Debug Output window. Naturally, you'll have to have the Debug Output window open in the Debugger for the values to be visible. Because I use the Debug Output window all the time, I usually have the Debugger Environment combo box in the Tools, Options dialog set to FoxPro Frame, and then I place the Debug Output window on the left side of the FoxPro desktop. See **Figure 19.16**.

I find the first expression to be a handy technique for marking the start of a set of output values. It identifies that a new run through the routine is starting, identifies which program is running, and notes the date and time of execution.



**Figure 19.15.** The Call Stack window shows you a list of form methods and events that have been executed to get to the current state.



**Figure 19.16.** The Debug Output window captures the results of the DEBUGOUT command.

### Debug Output window shortcut menu

Once there is content in the Debug Output window, you can right-click in the window and select the Save As menu option to select or name a file to which the data in the Debug Output window will be sent. This file is a plain text file and can be opened later in the Visual FoxPro editor or any other text editor.

You can also select the Clear menu option to get rid of everything in the Debug Output window—just like you do with the Command window.

**Docking View**

If this menu command is checked, you can dock the window inside the debugger frame. If this menu option is not checked, you won't be able to dock the window.

**Hide**

Simply hides the Debug Output window.

**Font**

Opens the Font dialog box so you can change the font name, style, and size.



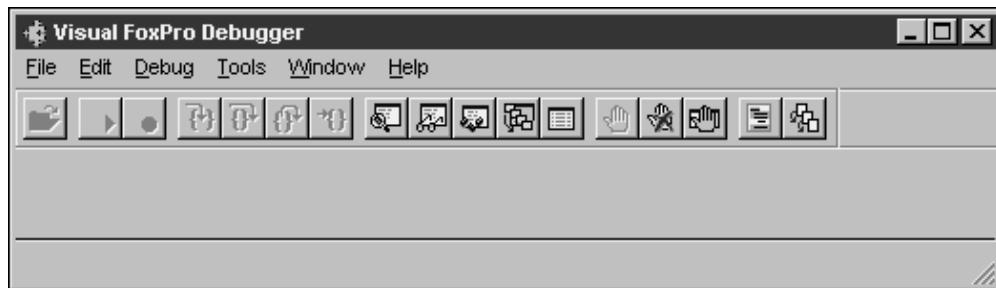
*The name of the Debug Output window is "Debug Output"—you might find this handy if you are trying to work with it programmatically. You need to enclose the name in quotes because of the space in the name. For example, to open the window with the ACTIVATE WINDOW command, use the statement*

```
activate window "Debug Output"
```

**Debugger toolbar**

I've mentioned bits and pieces of the Debugger toolbar before—it's time to discuss what each button does. But before I do, I should mention that the Debugger toolbar is available regardless of which debugging environment you're using. If you've got the Debug Frame open, the Debugger toolbar shows up under the menu, as shown in **Figure 19.17**.

However, if you're opening debugging windows inside the FoxPro Frame, you can still get to the Debugging toolbar. Open it through the Toolbars dialog, accessible from the View, Toolbars menu option, or right-click in the gray area of a toolbar and select the toolbar from the context menu that appears.



**Figure 19.17.** The Debugger toolbar.

The Debugger toolbar contains the following buttons from left to right:

**Open**

Performs the same function as the Open menu option in the Trace window shortcut menu.

**Resume**

Performs the same function as the Resume menu option in the Trace window shortcut menu.

**Cancel**

Cancels the currently executing program, and, in the Visual FoxPro interactive environment, returns control to the Command window.

**Step Into**

Performs the same function as the Step Into menu option in the Trace window shortcut menu.

**Step Over**

Performs the same function as the Step Over menu option in the Trace window shortcut menu.

**Step Out**

Performs the same function as the Step Out menu option in the Trace window shortcut menu.

**Run To Cursor**

Performs the same function as the Run To Cursor menu option in the Trace window shortcut menu.

**Trace window**

Opens or closes the Trace window.

**Watch window**

Opens or closes the Watch window.

**Locals window**

Opens or closes the Locals window.

**Call Stack window**

Opens or closes the Call Stack window.

**Debug Output window**

Opens or closes the Debug Output window.

**Toggle breakpoint**

Sets a breakpoint on or off.

**Clear all breakpoints**

Sets all breakpoints off.

**Breakpoints dialog**

Opens the Breakpoints dialog.

**Toggle coverage logging**

Starts or stops coverage logging. See Chapter 21 on the Coverage Profiler for more details.

**Toggle event logging**

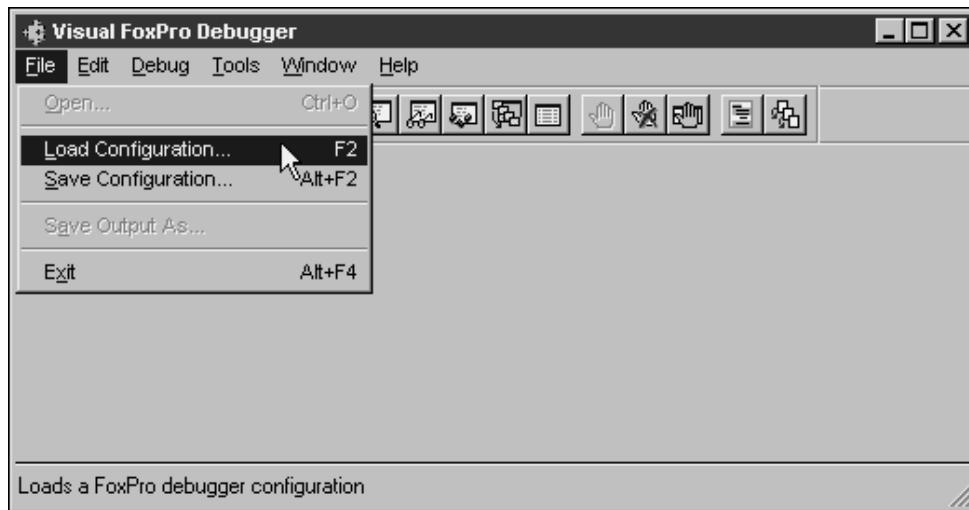
Starts or stops event logging. See the section on event tracking in this chapter.

**Debug Frame menu**

If you open the Debugger in the Debug Frame, the window has its own menu. Most of the options are available from the Debugger toolbar as well, but there are a couple of special features.

**File**

The File menu provides options for handling external files having to do with the Debugger, as shown in **Figure 19.18**.



**Figure 19.18.** The Debugger File menu.

**Open**

Performs the same function as the Open menu option in the Trace window shortcut menu.

**Load Configuration**

Loads a previously saved group of Debugger settings. See “Save Configuration” for details.

**Save Configuration**

You often might find yourself switching between one group of settings and another, and it's a real pain to have to keep entering the breakpoints, expressions, and so on each time. You can

save a particular configuration of the Debugger, including breakpoints, watch expressions, and events to be tracked. This information is saved through the Save Configuration menu option, and is stored in a text file with a .DBG extension. You can use the Load Configuration menu option to later reload a configuration that you previously saved.

### **Save Output As**

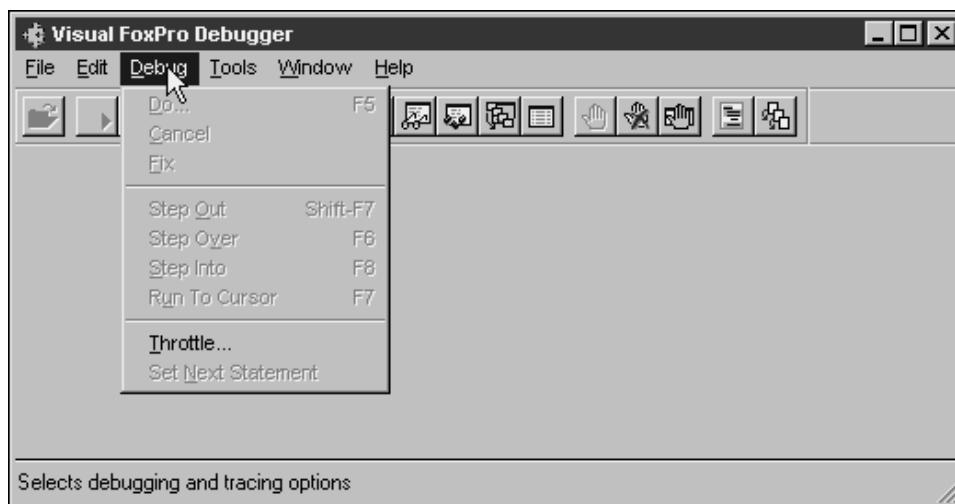
Performs the same function as the Save As menu option in the Debug Output context menu. This menu option is enabled only when there is content in the Debug Output window.

### **Edit**

The Edit menu contains the standard Cut, Copy, Paste, Select All, Find, and Find Again menu options that you would expect.

### **Debug**

The Debug menu provides access to functions used during the debugging process, as shown in **Figure 19.19**.



**Figure 19.19.** The Debugger toolbar's Debug menu.

### **Do**

Executes a program.

### **Cancel**

Cancels the currently executing program, and, in the Visual FoxPro interactive environment, returns control to the Command window.

**Fix**

Suspends program execution and allows source code to be modified.

**Step Out**

Performs the same function as the Step Out menu option in the Trace window shortcut menu.

**Step Over**

Performs the same function as the Step Over menu option in the Trace window shortcut menu.

**Step Into**

Performs the same function as the Step Into menu option in the Trace window shortcut menu.

**Run To Cursor**

Performs the same function as the Run To Cursor menu option in the Trace window shortcut menu.

**Throttle**

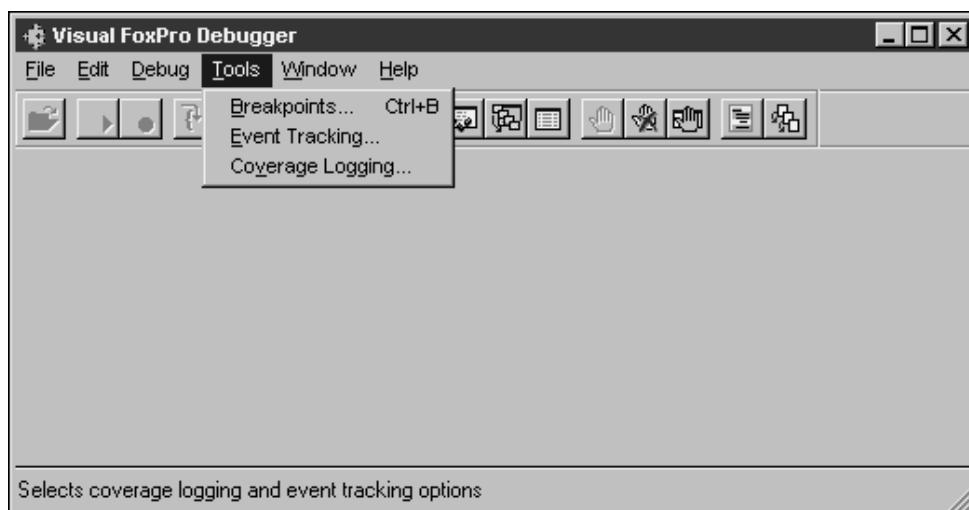
Opens the Execution Throttle dialog so that you can pause between executing statements.

**Set Next Statement**

Performs the same function as the Set Next Statement in the Trace window context menu.

**Tools**

The Tools menu provides access to several debugging tools as shown in **Figure 19.20**.



**Figure 19.20.** The Debugger toolbar's Tools menu.

**Breakpoints**

Opens the Breakpoints dialog.

**Event Tracking**

Opens the Event Tracking dialog. See the section on event tracking later in this chapter.

**Coverage Logging**

Opens the Coverage Logging dialog. See Chapter 21 on the Coverage Profiler.

**Window, Help**

The Window and Help menus both contain the standard menu options you would expect. The only difference is that the Window menu has hard-coded menu options for each of the five Debugging windows.

## Event tracking

Once you start building forms that are more complex than just a bunch of controls mapped to a single table, you'll find yourself starting to wonder if the error you're running into is a result of one method firing at an inappropriate time with respect to another. To make sure, many developers put markers of one sort or another in specific events that they are interested in. These markers can take the form of message boxes, WAIT WINDOW statements, or even DEBUGOUT statements. Everyone has written code like this:

```
messagebox("We are starting the Init() method of the Customer form.")
```

or

```
WAIT WINDOW "The IMPORT process in the cmdPost button's " ;  
+ "Click method is about to begin"
```

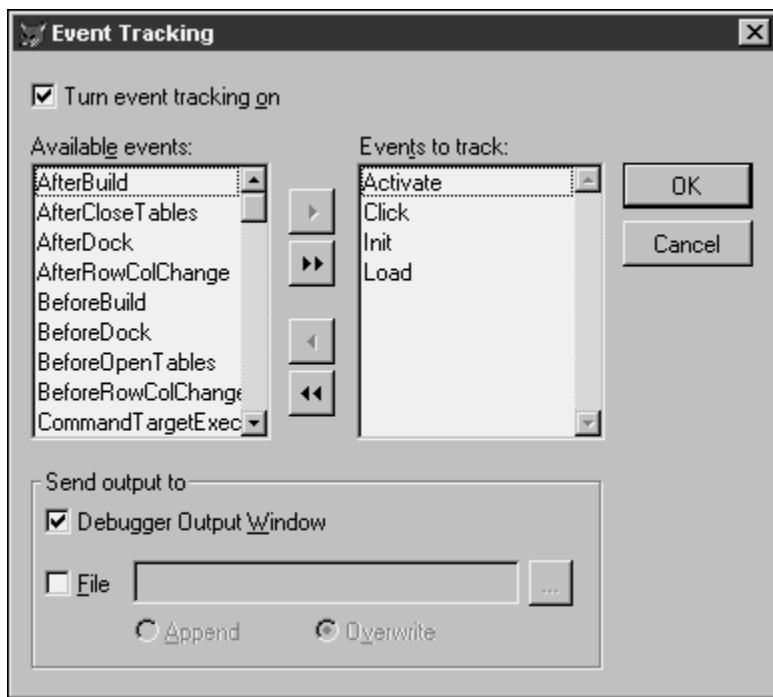
Both message boxes and WAIT WINDOW statements, while useful, are a real pain to deal with because (1) you have to click the mouse or press the Enter key for each marker, and (2) you have to remember to comment out or delete each of these lines before shipping your app to the user.

The DEBUGOUT statement is a bit more forgiving, because the user doesn't have a Debug Output window. But a lot of statements still garbage up your code; they might impose a performance penalty depending on where they are located; and heaven forbid if anyone else sees your application with this temporary code still in it!

Visual FoxPro now comes with a new tool, accessible through the Debugger, that allows you to send a listing of which events are firing, and in which order, to the Debug Output window or to a text file. This tool is called the Event Tracker.

## Using the Event Tracker

The Event Tracker can be started two ways. If you've not used it before, you should probably open it through the Tools, Event Tracking menu option in the Debugging menu. The dialog is shown in **Figure 19.21**.



**Figure 19.21.** The Event Tracking dialog.

The Event Tracking dialog contains two list boxes—one with a list of all available events in Visual FoxPro, and another that starts out empty. You'll use this dialog to identify which of the available events you want to track.

To get started, you'll first want to check the Turn event tracking on check box—if you don't, anything else you do won't matter. Next, you'll move events that you want to track from the Available events list box to the Events to track list box. You can move all of the events over, but that's probably a bad idea because an event record is generated each time one of the selected events fires. Some events fire only at specific times, such as Load() or Init(), but other events fire incessantly, such as MouseMove() and Paint()—you can slow down your application's performance to a crawl and fill up your event-tracking log quickly if you decide to track the MouseMove() event and then go crazy with your rodent.

After you've selected the events that you're interested in, decide whether you want to send the output to the Debug Output window or to a file. If you're not sure, send your output to the Debug Output window—you can always send the contents of the Debug Output window to a file later. Note that if you decide to send the results to a file, you can enter your own file name or pick an existing file, and you can choose to append or overwrite the contents of that file.

When you're done with this dialog, open the Debug Output window if that's the route you chose, and just run your application. When you're done with the section of the app that you're examining, flip over to the Debug Output window (or open the file) and examine the output.

Once you've set the events you want to track, you can turn event tracking on and off by clicking the Toggle Event Tracking button on the Debugging toolbar. This will prevent you from generating a 300K file before you ever get to the routine or section of code you're interested in.

You'll very likely find the Event Tracker useful in at least two specific situations. The first is when you have a complex set of interactions between controls, including list boxes and grids. It's very easy to make an incorrect assumption as to which events are firing in what order. The other is when you are seeing behavior that makes you believe a section of code isn't firing. You can track the specific events to verify what is and what is not being fired.

## **Going about the debugging process**

Now that you've started to really write some code, you are probably running into problems. In other words, things aren't going as planned or as expected. This is the time that weeds out the wannabes from programming. And it's all because of expectations.

Expectations are a funny thing. When there's a difference between what is happening on the computer and what you thought would happen, there's a difference between expectations and reality. Computers are different from humans, however. A difference in expectations between two people can often be resolved through compromise. With your computer, however, no compromise is possible—you have to do it the computer's way, period! But it's not always easy to find that way. Like a child, a computer can't always tell you "where it hurts;" the information it provides can be incomplete or misleading; and, most importantly, it's not always clear what you have to do to make it better.

Thus, when you're faced with computer misbehavior—specifically, when you're getting unexpected results—you have a challenge in front of you. The purpose of this chapter is not only to provide information on how to use the tools you have at your disposal to solve those problems, but, even better, to provide a philosophy and techniques for using those tools properly.

## **Types of misbehavior**

I'm going to assume that you're running an application, and that your problems are occurring as a result of using the various components in VFP—that you're not just trying to put a control on a form or set a property a certain way. There are five general types of errors you will encounter: defects in the product, compile-time errors, run-time errors, logic errors, and user-generated errors.

### **Defects in the product**

Visual FoxPro is in its third major revision, and it's been used by literally millions of people around the world. After this period of time, most of the defects have been found and corrected. Service Pack 3 for version 6.0 took care of more than 300 defects that had dated back to version 3.0. Sure, there are probably a few left, but it's highly unlikely that the issue you're facing is a defect in the product. It's much more probable that the problem lies elsewhere. Nonetheless, it's good to be aware that, when all else fails—and I mean really fails—the behavior you're experiencing could possibly be a bug in VFP itself. But we'll leave that possibility for the very last.

## Compile-time errors

A compile-time error is a programming error that you catch when compiling your program. For example, if you use the FOR clause in a SQL SELECT command and then try to compile the form, class, or program that contained that SELECT command, VFP will generate an error message in a Cancel/Ignore/Ignore All dialog:

```
Command contains unrecognized phrase/keyword.
```

There's a truism in our industry that has analogies in other businesses as well: The earlier a defect (or a problem, or a design defect, or an illness) is caught, the easier it will be to fix, and the less it will cost to do so. By catching errors almost at the same time you're creating the error, VFP is reducing your time and cost to fix it to almost nothing.

The only problem might occur when you can't figure out what the error is. In those cases, it can be quite a nuisance—all you want to do is save the darn form and run to the bathroom, but VFP keeps complaining. You can select Ignore to let the compiler continue, while ignoring that line, or select Ignore All to ignore that line and all future compile errors.

## Run-time errors

A run-time error occurs once you run the program. For example, suppose you are tracking the dollar value of automobiles owned by individuals attending a car show. To do so, you would sum up the total value of the automobiles, and then divide by the number of people at the event. However, if your program did this calculation before anyone was registered for the event, the calculation performed would be zero divided by zero.

As we all know, one mustn't divide by zero or else you'll have to sit in the corner for the rest of the day. When a computer divides by zero, an error will be generated. However, since the application didn't know it was going to be dividing by zero when the program was compiled, the error passed through the compiler, and was not caught until the program was run and those two statements were executed. Thus, it's called a "run-time" error.

## Logic errors

A logic error is one that violates the intent—or the business rules—that the program needs to use in order to function properly. For example, suppose you have a CASE structure that calculates dues based on which season the member joined the club. The CASE structure could look something like this:

```
do case
case thisform.txtSeason.value = "Winter"
  * they get charged a full year of dues
  nDues = thisform.txtDuesAnnual.value
case thisform.txtSeason.value = "Spring"
  * they get charged for nine months of dues
  nDues = thisform.txtDuesAnnual.value * 0.75
case thisform.txtSeason.value = "Summer"
  * they get charged for six months of dues
  nDues = thisform.txtDuesAnnual.value * 0.5
```

```
case thisform.txtSeason.value = "Fall"
  * they get charged for three months of dues
  nDues = thisform.txtDuesAnnual.value * 0.25
endcase
* multiply dues by number of cars
nDues = nDues * thisform.txtNumberOfCars.value
```

What's the problem? First, if the user could enter the season via a text box that allowed any type of data (admittedly, a bad choice of controls), they might end up entering a value like "Autumn," or misspelling a season, like "Witner." Suppose, however, you provided a four-button option group instead, and the user could choose only one of those four choices. What if they didn't make a choice? The value of txtSeason (it would probably be called opgSeason, actually) would be empty.

In any of these cases, there wouldn't be a value assigned to the variable nDues, and then the next line—in which the number of cars is factored into the dues—would fail. This is an example of a logic error that then generates a run-time error, but it doesn't tell you the location of the logic error, making it tough to track down. There are two ways to correct this particular problem. The first is to initialize the value of nDues before the CASE statement, like so:

```
nDues = 0.00
```

By doing so, the statement where the number of cars is factored into the dues wouldn't cause a run-time error. However, it still might not produce the correct answer. The second way, and the better alternative, is to include an OTHERWISE statement in which you handle the possibility that none of your CASE statements account for the actual value of thisform.txtSeason.value. For example, the following code placed before the ENDCASE statement would work fine:

```
otherwise
  nDues = 10.00
```

Another example of a logic error is when you haven't correctly analyzed the rules for the application. You're performing the operation correctly—it's just that you're performing the *wrong* operation correctly. For example, suppose the club rule stated that a member was charged the minimum annual amount for dues—regardless of how many cars they owned, or even if they didn't own any—and then an additional amount was included for each automobile they owned. The code might look like this:

```
* multiply dues factor by number of cars
nDues = (nDues + nDuesFactor) * thisform.txtNumberOfCars.value
```

The parentheses indicate that the Dues and the DuesFactor are added together, and the result multiplied by the number of cars. This would produce the wrong result for two reasons. First of all, the DuesFactor was an additional value that would be multiplied by the number of cars—and that result would be added to the Dues calculated in the CASE structure. Secondly, the intent was to have a base amount that every member was charged, regardless of how many cars they owned. The equation above would result in a dues amount of \$0.00 if they

owned no cars. The statement wouldn't throw an error if thisform.txtNumberOfCars.value was equal to zero, but it wouldn't calculate the correct amount, either. Again, a logic error. Logic errors are usually the nastiest kind, because there's often no error shown to the user—only aberrant behavior that may or may not manifest itself in an obvious fashion. If it doesn't—if the error occurs only when the 13th of the month is on a Friday—you could go months before you run into it, and then more months before it happens again. And who knows how long it would take you find the cause?

### User-generated errors

The last type of error is that which cannot be controlled by the developer—an action by the user outside the realm of the application. For example, suppose the user is printing a report, and turns the printer off partway through, or the printer runs out of paper. Or what if your application is set up to use a separate “library” file, but some unwitting user, trying to be helpful and clean up some disk space, has deleted that file, not knowing what it was? The application will fail as soon as a function from the library is called anywhere in the application. Or perhaps the user is exporting data from the application or importing from another source, and during the operation, another user on the network copies a big file onto the same drive, using up all the disk space.

While these types of errors are not your fault, you will have to consider the possibility that some will occur, as appropriate for your application. (Obviously, if you don't ever let your users print, then you wouldn't have to worry about printer errors, for example.) You'll have to plan your application in order to detect errors and react to them so that the application doesn't simply crash or stop operating.

### Debugging covers developer-generated errors

The following discussion will cover the middle three errors. You are, in reality, at the mercy of Microsoft when it comes to those rare defects in VFP itself. And while it is incumbent upon you to handle user-generated errors as well, that should be taken care of via an error handler in your code, which is clearly outside the domain of debugging.

### The debugging mindset

The typical programmer—in other words, the self-taught amateur who lacks discipline, rigor, and an interest in achieving those attributes—will make a series of random, uncoordinated attacks on various parts of the application, basing those attacks on wild guesses and perceived ease of implementation. In other words, if they succeed, it's simply because they're lucky. A great many of these programmers make their way through their careers, narrowly escaping great danger as they get lucky one time after another.

This isn't uncommon, because many problems—not all—can be solved through such a haphazard method of trial and error. However, when it doesn't work, the aforementioned programmer must resort to any number of tricks, including simply accepting an error that doesn't occur often enough to fix, unnecessarily rewriting pieces of code that were 99% finished, and telling the customer, in error, “It can't be done” or “There's a bug in Access that prevents me from making this work.”

There is a better way, obviously, or I wouldn't be spending so much time berating the “wild, random guesses” methodology I just described. It is important to develop a “debugging

mindset” where you approach an issue of errant misbehavior in a logical, rational manner, much as you learned about the scientific method in seventh grade.

### ***Make observations***

The first action is to describe the errant behavior. There are four parts to this description:

- Part 1 is a list of steps to reproduce the behavior. If you can’t reproduce the behavior, then you might as well put it down as “one of those things” and move on. This entire process depends on being able to duplicate the behavior so you can analyze it. Furthermore, by writing down the steps, if it goes that far, you will often find that the behavior was caused by forgetting to perform a step, not by anything else.
- Part 2 is a description of what happens as a result of following the steps described in part 1. Sometimes this description gets intertwined in the steps, because there are several related behaviors, and each behavior generates an additional step on your part, which then causes the application to exhibit another behavior.
- Part 3—and this is the key—is to describe what you expected to happen. By doing so, you make the difference in expectations very clear: Part 2 describes reality, and this part describes your intended result. The difference, then, is what needs to be investigated. Again, like in part 2, actually thinking through (or even writing down) your expectations will often generate the “Aha!” moment, whereby you realize that your expectation was misguided. It might become obvious that an entirely different behavior should really have occurred.
- Part 4 in this first step is to list any “other interesting facts” that you think might have bearing on the problem at hand.

### ***Go through the scientific method***

Now that you have a description of the problem, it’s time to figure out what is going wrong.

1. Generate hypotheses.

In this step, you’ll create one or more hypotheses, and then test each hypothesis. Did a light bulb suddenly go on during this last sentence? If you’re thinking that this process maps closely to the traditional scientific method, you’re correct. Gather facts and observations, and then generate hypotheses about what might be the cause. You thought seventh-grade science would never be useful!

What could possibly be going wrong? Unfortunately, there isn’t any magic methodology for creating these hypotheses, which is why scientific geniuses aren’t a dime a dozen. It’s a special talent to be able to come up with “good” hypotheses. You can begin by tracing the code to determine whether you can generate possible problems based on what you see. Perhaps you aren’t using a function properly. Perhaps the function doesn’t return the type of value you think it does. Perhaps you’re assuming that a variable contains a certain value when, in fact, it contains a different one. And so on and so forth.

## 2. Test your hypotheses.

Now that you have one or more hypotheses, it's time to test them. For example, suppose you suspect that a function is returning a value other than the one you think it is. Don't wait until the end of the routine to find out the answer you're getting isn't correct—find out that value right away!

## 3. Repeat.

This is the tough step. If your hypothesis is correct, you're basically done. Make the appropriate change to the code, or figure out what else you need to do in order to fix the problem, and then test it again. If, however, your hypothesis was not correct, you need to repeat the prior two steps—again and again until you find the answer.

I can't stress this enough: You must repeat these steps until you find the answer. Start at the very beginning and examine every assumption—including the ones you know to be true. Check each fact, and check it again: Check your observations and do everything step by step. Most problems occur at this stage for one of two reasons. Either you're rushing through and making an incorrect assumption, or you don't understand how the product actually works, which, really, is another case of making an incorrect assumption.

### **Know why it worked**

Simply fixing a defect isn't enough, though. It's kind of like whacking your TV upside the tuner every time the picture goes fuzzy. It might seem like you've fixed it, when all you've done is shake a bad tube partway out of its seating. In a few minutes, as the temperature in the case rises, the tube will fall back into place, and the picture will go fuzzy again. Wouldn't it be better to just get the tube replaced and fix the root of the problem? (Or, as my tech editor suggested, perhaps you should just go replace your 40-year-old TV with one that doesn't use tubes. I guess I was thinking about the *picture* tube.)

Same thing here—only more so. If you perform an action that “seems” to fix a bug, you can pretty much count it appearing somewhere else. You might have found a clever workaround for the particular situation, but you've not learned anything—thus, you're bound to make the same mistake somewhere else, perhaps where it's not going to be as easy to work around.

And don't relax if you run into one of those bugs where “it just went away on its own.” As has been said many times, “Bugs that go away on their own tend to come back by themselves.” Steve McConnell, in his landmark programming text, *Code Complete* (Microsoft Press), says “If you aren't learning anything, then you're just goofing around.”

### **Find (and fix) one thing at a time**

Finally, resist the temptation to fix a bunch of bugs at the same time, and then run the application to see if you've fixed them all. This technique is tempting because it seems to be much more efficient. Fixing the defects one at a time seems an awful lot like going to the car dealership to get the muffler fixed, picking it up and returning it later that day to get the

window repaired, and then picking it up and returning it yet again to have the engine tuned. Why not have it all done at once?

Two reasons, folks. The first answer is that you're simply not going to remember everything to fix, or if you did, you won't remember everything to test afterwards. There are just too many other things that can intrude—and your concentration is much more apt to be interrupted if you try to juggle a chainsaw, a flaming torch, and a balloon full of sharp glass than if you handle one at a time. The other reason is that one fix might improperly affect the other. If you fix one defect and test it until the issue is resolved, you can put it to bed, knowing the "whats" and "whys." If you fix two things—in other words, if you change two things—at the same time, you won't necessarily know which change affected which problem (or possibly both problems). As I've said before, "A man with a watch knows what time it is. A man with two is never quite sure."

## A final word

Now that you've read all this, it might seem that each bug you run into, if you approach it properly, will take the better part of an hour to squash. That's not necessarily so, although sometimes, sadly, it is. Many times you can perform all of these steps mentally, and in a fraction of a minute. You don't necessarily have to write out "Steps to reproduce" or "Expected Behavior." You don't have to formally document each hypothesis when you run into an "Invalid Value" error message on a statement that has only two variables.

However, just because many bugs can be detected and disposed of quickly doesn't mean that you can always do so. And when the bugs get tough, it's time to introduce more rigor and formality into your bug-stomping process. Now you have a guideline for doing so.

# Chapter 20

## Builders

**As Visual FoxPro becomes more and more mature, the development team correspondingly has to spend less and less time creating core features, and more and more time enhancing the productivity features of the product. One often-overlooked productivity feature of Visual FoxPro is its Builder technology. In this chapter, I'll introduce you to Visual FoxPro Builders, and show you how to use the technology to build your own builders as well.**

There are two mechanisms in Visual FoxPro that are supposed to assist you with various common tasks: wizards and builders. But what's the difference?

A wizard is a tool that guides you through the creation of a table, a form, a report, or some other process. Suppose that you've never produced a report in Visual FoxPro, but you need to create one. The Report Wizard will ask you questions about the type of report you want and then create one for you. I use wizards with software that I don't use very often, or those that I'm just starting to learn. For example, I use the wizards in PowerPoint about twice a month, because that's about how often I use PowerPoint. I'm simply not familiar enough with the product to be able to use it skillfully myself, and my demands aren't that sophisticated, so I let the wizards do the work for me.

Some wizards automate a series of steps that you might do often, but that require the meshing of disparate technologies or have a lot of standard, behind-the-scenes automation involved with them. The Visual FoxPro Setup Wizard is a good example—it leads you through the steps required to create a set of files that you can distribute to your users.

Wizards are good for getting started with a product, but they aren't terribly sophisticated. I don't mean that they're not complex; it's just that a wizard allows you to do what *it* wants you to do—and nothing more. What if you had a wizard that you could tailor for your own purposes? What if you could create your own wizards? What if you could make a wizard access a database to determine what it should do, instead of relying on hard-coded instructions?

That's what a builder is—sort of a customizable wizard. Sure, the name sounds like a blue-collar version of a wizard, but it's really a completely different breed of cat. You know that funny feeling you get when you try to answer the question, "What kind of programs can you write using Visual FoxPro?" Same thing with builders—it's difficult to pin down just what a builder can do. In short, a builder is a mechanism for creating and modifying forms and controls. Visual FoxPro comes with a series of builders that help you manipulate forms, format and modify some of the standard controls, and create referential integrity between tables.

However, the beauty of the Visual FoxPro Builder technology is that you aren't limited to the builders that come with the product—you can create your own and call them at selected times, instead of calling Visual FoxPro's built-in builders. In fact, one of the members of our community, Ken Levy, has extended the builder technology. His tool, BuilderD, is a data-driven set of classes that allows you to create builders that create builders. If your head is starting to spin, that's all right—just sit down and think of a robot factory whose main product is more robots.

This chapter will explain how the Visual FoxPro Builder mechanism works, and how to create your own builders. Finally, I'll create three builders that will demonstrate the breadth of what you can do with a builder, and show you where to go next if the BuilderD technology sounds intriguing.

## **The Visual FoxPro Builder technology**

Think about what happens when you create a form. First, after issuing the CREATE FORM command, you place a control on the form. Then you open the Properties window and set a number of properties. Commonly, these properties include the name, the size (height and width), and a caption, as well as specific properties according to the type of control, such as the data source, how the control is initially populated, and so on. Yup, for each property you want to set, you either scroll through the list of properties in the All tab, or try to remember if the TabIndex property is on the Layout or Other tab. Once you've finished with the control, you add a second control and go through the same process. And again for the third control, and the fourth, and, yes, this gets tedious in short order. This isn't the best use of your time, is it? It's the programmer's version of data entry, and when you think about it, that's exactly what you're doing—data entry into the Properties window for each control.

Wouldn't it be nice if you had a "quick properties" tool that allowed you to set the half-dozen properties that you always used for a specific control? Perhaps a data-entry form that asked you what the name of the control should be, and what the caption should be? And perhaps you could set up a series of three or four standard sizes for the control, and then just select from that list, instead of trying to remember "Did I make these command buttons six pixels apart or eight?"

Visual FoxPro Builders provide this type of interface to the data-entry chore of entering values in the Properties window. How about trying one out. Create a form and place a text box on the form using the Form Controls toolbar. Then, right-click the control to bring up its shortcut menu. Select the Builder menu option, and the Text Box Builder will appear as shown in **Figure 20.1**.

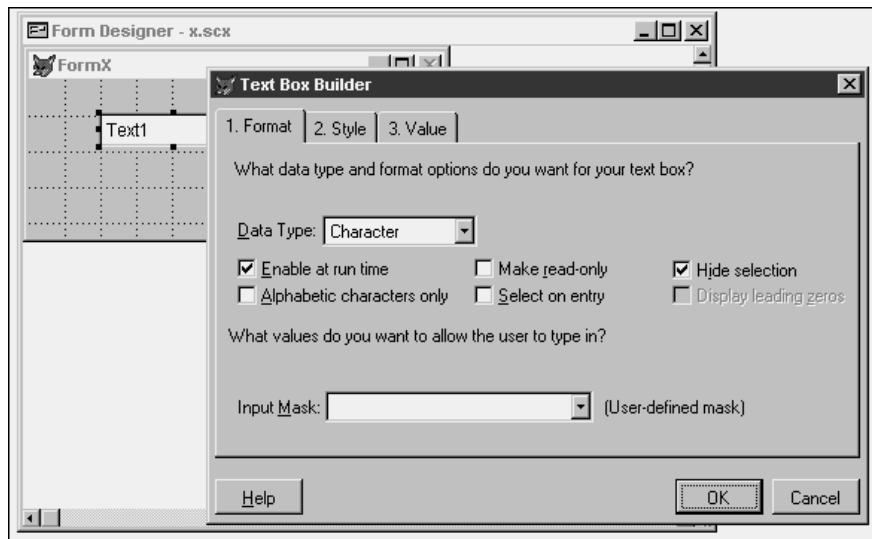
This dialog allows you to set the format, style, and control source for the text box you've just placed on the form. And here's the fundamental idea behind a builder: *Each of the choices—each control on each tab in the builder—maps to a specific property for the text box control.* The control will change on the fly as you select options. This enables you to see the results of your changes before you finish with the builder.

This wouldn't be Visual FoxPro if there weren't 20 or 30 different ways of doing the same thing. You can also bring builders forward automatically by selecting the Builder Lock icon on the Form Controls toolbar, and then placing controls on the form. As each control is placed on the form, the appropriate builder will appear for you to do with what you wish. Additionally, you can keep the Builder Lock icon selected by default by selecting the Builder Lock check box in the Forms tab of the Tools, Options dialog box.

There are several pieces that play nicely together in order to make this work.

The first is the \_BUILDER system variable. It is set to BUILDER.APP, located in the Visual FoxPro directory (you can see this by looking at the File Locations tab in the Tools, Options dialog box). When you select the Builder menu option or use the Builder Lock button in the Form Controls toolbar, the program or application specified by this system variable is automatically run.

BUILDER.APP does a whole bunch of stuff, but you can think of it as a “Builder Traffic Cop” whose main responsibility is to figure out which builder to run. Whoa! “Which” builder to run? There are more than just the native Visual FoxPro builders, like the Text Box Builder shown in Figure 20.1? You betcha!



**Figure 20.1.** Open the Text Box Builder by selecting the Builder menu option from the control’s shortcut menu or by selecting the Builder Lock icon from the Form Controls toolbar before placing the control on the form.

The second piece you need is a table called BUILDER.DBF, which is located in the HOME() + \WIZARDS directory. It contains a list of the default Visual FoxPro builders, as shown in **Figure 20.2**.

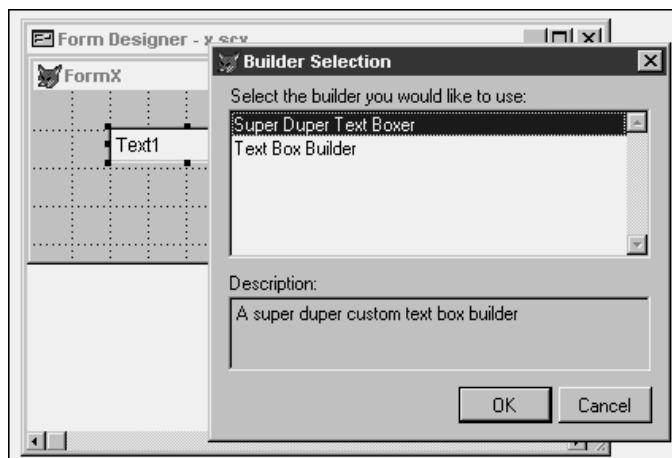
You can add your own builder definitions to this table as well. If you do so (I’ll explain how shortly), clicking the Builder menu option will display a list of available builders for that type of control, as shown in **Figure 20.3**. Of course, BUILDER.APP does all this automatically. But there’s more!

You can add a property called BUILDER to a control’s base class. Then you can store the name of a builder to be used for that control in the BUILDER property. When you select the Builder menu option from the control’s context menu, that builder will automatically run. (I’ll discuss how to do this shortly as well.) Note that if there are other builders registered in BUILDER.DBF for that type of control, those builders will be ignored—you won’t get a pick list of the registered builders in addition to the BUILDER property builder.

You could even write your own replacement for BUILDER.APP, and reference your new builder application in Tools, Options if you wanted. However, you’re likely to find that you can get an awful lot done simply by creating builders that are called by BUILDER.APP, so writing your own builder app is an exercise left to the reader.

| Name                          | Descript | Bitmap | Type         | Program | Classlib | Classname | Parms |
|-------------------------------|----------|--------|--------------|---------|----------|-----------|-------|
| Edit Box Builder              | Memo     | Memo   | EDITBOX      | Memo    | Memo     | Memo      | memo  |
| Option Group Builder          | Memo     | Memo   | OPTIONGROUP  | Memo    | Memo     | Memo      | memo  |
| List Box Builder              | Memo     | Memo   | LISTBOX      | Memo    | Memo     | Memo      | memo  |
| Grid Builder                  | Memo     | Memo   | GRID         | Memo    | Memo     | Memo      | memo  |
| Form Builder                  | Memo     | Memo   | FORM         | Memo    | Memo     | Memo      | memo  |
| Combo Box Builder             | Memo     | Memo   | COMBOBOX     | Memo    | Memo     | Memo      | memo  |
| Command Group Builder         | Memo     | Memo   | COMMANDGROUP | Memo    | Memo     | Memo      | memo  |
| AutoFormat Builder            | Memo     | Memo   | AUTOFORMAT   | Memo    | Memo     | Memo      | memo  |
| Referential Integrity Builder | Memo     | Memo   | RI           | Memo    | Memo     | Memo      | memo  |
| Text Box Builder              | Memo     | Memo   | TEXTBOX      | Memo    | Memo     | Memo      | memo  |
| AutoFormat Builder            | Memo     | Memo   | MULTISELECT  | Memo    | Memo     | Memo      | memo  |

**Figure 20.2.** BUILDER.DBF contains a row for each builder that can be invoked by BUILDER.APP.



**Figure 20.3.** If BUILDER.DBF contains more than one row for each control type, a Builder Selection dialog will open when you select the Builder menu option from a control's context menu.

## Setting up your own builder in BUILDER.DBF

I'll use a very simple builder to demonstrate how to add your own builder records to BUILDER.DBF. This builder will do very little useful work—it will simply set the font name and font size of a group of controls to a hard-coded value. Yes, I know that you can do this now with the Multiple Selection tool, but it will be an easy demonstration:

1. Write the builder program. The program, called LASSO.PRG, is as follows:

```
* lasso.prg
*
* builder that sets the font name and font size of
* all selected controls in the form
*
*
* these parms are passed by BUILDER.APP but not used here
lparameters m.lcX, m.lcY, m.lcZ

local array laObjSel[1]
local lcFontName, lnFontSize, lnNumObj, i

*
* the font name and size are hard-coded for this example
*
lcFontName = "Courier New"
lnFontSize = 14

*
* place the references of all selected controls into an array
*
lnNumObj = aselobj(laObjSel)

*
* if we've got some in the array, change the
* font name and size for each control
*

if lnNumObj > 0
  for i = 1 to lnNumObj

    laObjSel[i].FontName = lcFontName
    laObjSel[i].FontSize = lnFontSize

  endfor
endif
```

There's a subtle command in this listing that's the key to the whole builder concept—the ASELOBJ function. ASELOBJ is one of those really smart functions: It places object references to currently selected controls in the active Form Designer window into an array and returns the number of items in the array. As a result, you now know which objects were lassoed, and you can address each one of them.

2. Create a new record in the BUILDER.DBF table. Remember, it's in HOME() + "\WIZARDS", so go ahead and open it like so:

```
use HOME() + "\WIZARDS\BUILDER.DBF"
```

3. Add a record with the following field values:

| Field    | Contents                                            |
|----------|-----------------------------------------------------|
| Name     | Lasso Builder                                       |
| Descript | Changes the Font Name and Size of multiple controls |
| Type     | MULTISELECT                                         |
| Program  | LASSO.PRG                                           |

Note that BUILDER.APP expects LASSO.PRG to be in the HOME() + “\WIZARDS” subdirectory—if you put it elsewhere, you’ll need to include the fully qualified path in the PROGRAM memo field.

Note also that you don’t have to create a program; if you wanted to use a class library, you could put the class library name (the .VCX) in the CLASSLIB memo field, and the name of the specific class that holds the builder in the CLASSNAME field.

4. The last step is to close the table and test the builder. To do so, create an empty form and place three text boxes on it. (The font name should be Arial and the font size should be 9 by default.) Then select two of the text boxes, right-click, and select the Builder menu option. You should get a Builder Selection dialog that includes the Lasso Builder as one of the choices. Select it, and you’ll see the font name and size of the two selected text boxes change.

## **Setting up your own builder in a BUILDER property**

But what if you know that you want to run a very special builder for a specific class? For example, suppose you have a particular text box class that displays a type of value in a certain way, and you want to change a whole raft of properties each time you drop that text box class on a form?

It’s more involved than the previous example, but here’s how you would go about it. In this example, I’m again going to change just the font name and font size of the text box:

1. Create a class for the builder to be run on the text box class. Create a custom class named TxtBld in a class library called BUILDERPROG.
2. In the Init() method of this custom class, add the following code:

```
* init()
local array laObjSel[1]
local lcFontName, lnFontSize, lnNumObj, i

*
* the font name and size are hard-coded for this example
*
lcFontName = "Courier New"
lnFontSize = 14

*
* place the names of all selected controls into an array
*
lnNumObj = aselobj(laObjSel)
```

```
*  
* if we've got some in the array, change the  
* font name and size for each control  
*  
  
if lnNumObj > 0  
    for i = 1 to lnNumObj  
  
        laObjSel[i].FontName = lcFontName  
        laObjSel[i].FontSize = lnFontSize  
  
    endfor  
endif
```

If this code looks familiar, good—it's identical to the LASSO program I used in the previous example.

3. Next, create a class for the control of interest. For example, create a text box class named txtBuilder in the class library BUILDERCTRL. Add a property named BUILDER to this new text box class, and enter this string in the BUILDER property for the class:

```
BUILDERPROG, TxtBld
```

4. Create a form, and add the txtBuilder text box to the form. (You'll need to add it to the Controls toolbar.)
5. Right-click on the text box and select the Builder menu option. The TxtBld class will execute, the text box font name will be changed to Courier New, and the font size will be changed to 14. That's all there is to it.

## Creating a user interface for a builder

The last example was all fine and good—as an example. But let's face it: You're not about to hard-code values in a program for a builder. The whole point is to allow the user of the builder (You? The developer down the hall?) to specify values after they run the builder themselves.

So you're probably wondering where you hook the user interface in.

It's fairly straightforward. Let's suppose you want to let the user pick the font size for the text box. All you need to do is pop up a form for the user to enter the new font size, and return the value to the program calling the form. There is a bit of extra plumbing involved, all of which has to do with certain assumptions that BUILDER.APP makes.

1. BUILDER.APP expects to see a form class, and will try to pass three parameters to it, even if you don't use them. Thus, the Init() method needs an LPARAMETERS statement, like so:

```
* init()  
lparameter dummy1, dummy2, dummy3
```

2. Next, the builder assumes a number of built-in properties and methods, namely lAutoShow and lAutoRelease, and Show() and Release(). (In fact, BUILDER.APP doesn't even check to see if they exist!) You can fill the Show() method with all of the code that was in the Init(), like so:

```
* show()
LPARAMETERS nStyle
local array laObjSel[1]
local lcFontName, lnFontSize, lnNumObj

*
* the font name and size are hard-coded for this example
*
lcFontName = "Courier New"

do form GETXTBLD to m.lnFontSize

*
* place the names of all selected controls into an array
*
lnNumObj = aselobj(laObjSel)

*
* if we've got some in the array, change the
* font name and size for each control
*

if lnNumObj > 0
  for i = 1 to lnNumObj

    laObjSel[i].FontName = lcFontName
    laObjSel[i].FontSize = lnFontSize

  endfor
endif
```

3. Then add this code to the Release() method:

```
* release()
release This
```

4. You would then create a form named GETXTBLD, including a text box for the user to enter their choice of font size, and a property to store the value the user entered. You might also have a command button with code like this in it:

```
thisform.uRetVal = thisform.text1.value
thisform.release
```

Naturally, you'd make this form modal, and you'd have code like this in the Unload event, just like I discussed in Chapters 14 and 15:

```
m.luRetVal = thisform.uRetVal
return m.luRetVal
```

When the builder runs, the form appears, the user enters a value, and that value is returned to the m.lnFontSize variable in the builder program.

Of course, your own builders will get more complex in short order, but this is the guts of how this mechanism works.

## Data-driving your builder-building process: BUILDERD

If you open the FoxPro Foundation Classes that come with Visual FoxPro 6.0, you'll see that all classes have two custom properties named BUILDER and BUILDERX. Furthermore, the BUILDERX property is set to the following in each class:

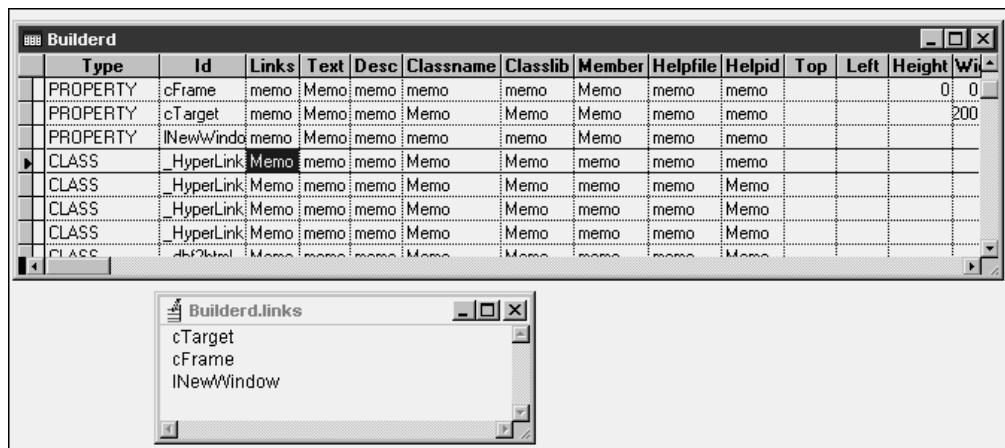
```
=HOME() + "\WIZARDS\BUILDERD.BUILDERDFORM"
```

Welcome to BuilderD.

Back in the days of Visual FoxPro 3.0, a fellow named Ken Levy created a tool called "BuilderB"—short for "Builder Builder." The idea behind BuilderB was to automate the creation of builders, because creating a builder is actually like creating a mini-application—typically you'd have a data-entry form (with one or more pages) so that you could enter data that customized whatever you were building. When you think about it, each of these builders—data-entry forms, right?—is nearly identical. You do the same types of things over and over again, namely setting values for a fixed number of properties. This just kinda screams out for automation, doesn't it?

Well, along comes Visual FoxPro 6.0, and Ken decided he wasn't satisfied with the automation of builders. He decided to add data-driven-ness to the process, rewrote BuilderB, and voila! BuilderD ("D" stands for "Dynamic," not "Data-driven") was born.

Going into depth on how BuilderD works is beyond the scope of this chapter (or book), but to get you thinking along these lines, I'll show you what the BUILDERD.DBF table looks like in **Figure 20.4**.



The screenshot shows the FoxPro IDE interface. At the top, there's a title bar with the text "Builderd". Below it is a table window titled "Builderd" with 13 columns: Type, Id, Links, Text, Desc, Classname, Classlib, Member, Helpfile, Helpid, Top, Left, Height, and Width. The table contains several rows of data. Below the table is a smaller window titled "Builderd.links" which lists three entries: cTarget, cFrame, and INewWindow.

| Type     | Id         | Links | Text | Desc | Classname | Classlib | Member | Helpfile | Helpid | Top | Left | Height | Width |
|----------|------------|-------|------|------|-----------|----------|--------|----------|--------|-----|------|--------|-------|
| PROPERTY | cFrame     | memo  | Memo | memo | memo      |          | Memo   | memo     | memo   |     |      | 0      | 0     |
| PROPERTY | cTarget    | memo  | Memo | memo | Memo      |          | Memo   | memo     | memo   |     |      | 200    |       |
| PROPERTY | INewWindow | memo  | Memo | memo | memo      |          | Memo   | memo     | memo   |     |      |        |       |
| CLASS    | _HyperLink | Memo  | memo | memo | Memo      |          | Memo   | memo     | memo   |     |      |        |       |
| CLASS    | _HyperLink | Memo  | memo | memo | Memo      |          | Memo   | memo     | memo   |     |      |        |       |
| CLASS    | _HyperLink | Memo  | memo | memo | Memo      |          | Memo   | memo     | memo   |     |      |        |       |
| CLASS    | _HyperLink | Memo  | memo | memo | Memo      |          | Memo   | memo     | memo   |     |      |        |       |
| CLASS    | _HyperLink | Memo  | memo | memo | Memo      |          | Memo   | memo     | memo   |     |      |        |       |

**Figure 20.4.** The data structure of BUILDERD.DBF.

It might not be completely obvious, but this is just a set of commonly defined properties, and then definitions of classes that link to those properties. When you create a builder for, say, a Hyperlink class (you can see the various FoxPro Foundation Class \_HyperLink classes in Figure 20.4), the information in the main class record is linked to records with information about other classes that make up the main class, and to properties as well. This information is used to create controls on one or more pages of a page frame in the builder. The controls that get plopped on the builder's pages are defined through base classes in BUILDERD.VCX.

This means that you can store in this table all of the parameters you would commonly use to create a builder, and create your builders automatically.

Yeah, I know that's a lot to swallow considering that I just introduced builders a few pages ago, but this should give you a feel for the extensibility of VFP's builder technology. If you're interested in more information about BuilderD, Doug Hennig wrote a pair of articles in *FoxTalk* ("Building Builders with BuilderD," March 1999, and "Building a Builder Builder," April 1999), that go into great depth. The *FoxTalk* Web site is at [www.pinpub.com/foxtalk](http://www.pinpub.com/foxtalk).

In this chapter I've given you an introductory glimpse into the world of builders. While introduced in Visual FoxPro 3, builders are still a completely new technology to most FoxPro developers, and I've just scratched the surface in order to show you how they work. You'll want to spend some time working with this, helping yourself automate your own work as well as that of your users. Finally, the shoemaker's children will have shoes of their own.

# Chapter 21

## The Coverage Profiler

The rest of the tools and ideas covered in this section have had to do with writing applications. This chapter covers the only tool that helps you with the next part—testing. There are several key processes you can perform to improve the quality of your software. These are (1) code reviews, a one-on-one or group review of actual code, (2) automated testing of the functionality documented in the specification for the product, and (3) an analysis of which lines in an application were actually run during testing. The Coverage Profiler helps you with this third task. In addition, the Coverage Profiler provides information about how many times a line of code was run, and how long it took—valuable information you can use to improve the performance of your application.

Visual FoxPro's Coverage Profiler performs two separate functions for developers interested in improving the quality of their software by looking at what code has actually been run during testing.

The first of these, the Code Logging engine, actually creates a log file that holds all information about which lines are run, in what order, and how long each line takes to execute. However, by itself, this log file isn't very interesting. Rather, it's interesting, but it's also really, really big, and it's just about impossible to digest it through visual inspection. It's the type of data that just screams out for an automated tool to suck it in and spit back out a variety of reports that are much more readable.

To get to this point, however, you have to configure the Coverage Profiler to do what you want it to do.

### Quick start with the Coverage Profiler

You can configure the Coverage Profiler in a number of ways. I'll start off with the basics and then show you how to customize it.

To follow along, you'll need an application to run the Coverage Profiler against—you can use the sample applications in previous chapters (such as Chapter 15), or you can pick your own application if you like.

### Basic configuration

If you're like most developers, you probably started up the Coverage Profiler by selecting the Tools, Coverage Profiler menu option, and were immediately greeted by a message that required you to select an existing log file, as shown in **Figure 21.1**.

Unfortunately, because this is the first time you started the Coverage Profiler, you didn't have a log file, so you clicked Cancel and gave up on the Coverage Profiler in frustration. I agree that this is sort of a lame interface. What's happening is that you're running the reporting application instead of starting the logging engine. Here's how to really get started.

### **Step 1: Start the code logging engine**

First, you need to get the code logging engine started. You do this by specifying a log file that will hold all of the coverage profiling information. You can specify a log file in one of two ways: by using SET COVERAGE or by using the Debugger.

#### **Use SET COVERAGE**

Issue the following command in the Command window:

```
set coverage to MYLOG.LOG
```

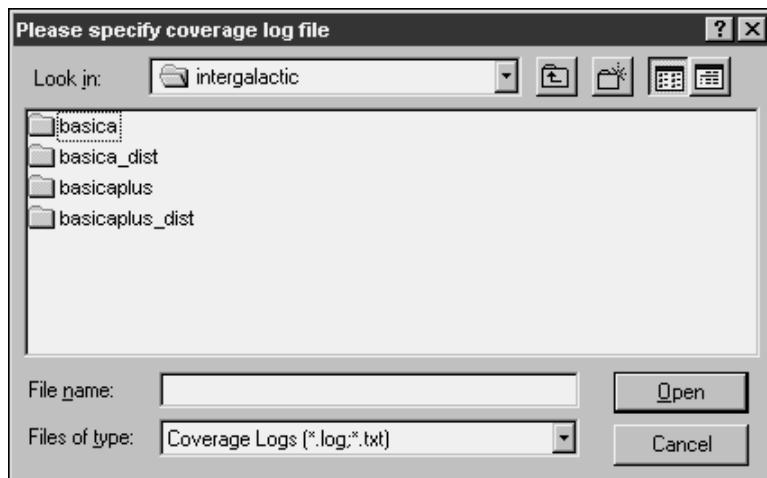
You can also do this in your code—perhaps bracketed by some “Developer Only” switch—if you don’t want to log every line from the start of your app.

#### **Use the Debugger**

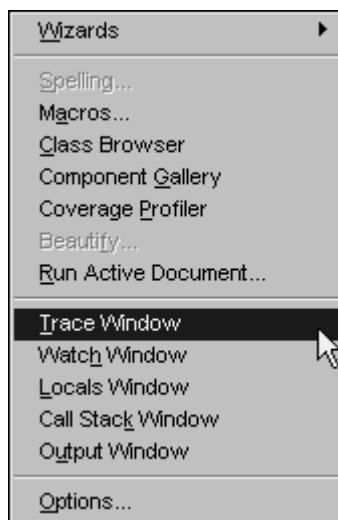
The second way to specify a log file is to use the Debugger. However, you can only get to the Coverage Profiler options in the Debugger if it’s set up in a separate frame, or through the Debugger toolbar if you have one of the five Debug windows open. You can tell whether you have the Debugger in the FoxPro frame or a separate frame by opening the Tools menu. If all five debug windows are shown in the Tools menu, as shown in **Figure 21.2**, the Debugger is set to a FoxPro frame.

To switch to a separate frame, open the Options dialog by selecting the Tools, Options menu option and select the Debug tab. Then select the Debug Frame option in the Environment combo box, as shown in **Figure 21.3**.

Now, there is only one menu option in the Tools menu—Debugger. When you select it, you’ll get a completely separate window—outside the Visual FoxPro application window as shown in **Figure 21.4**. See Chapter 20 on the Debugger for more information.



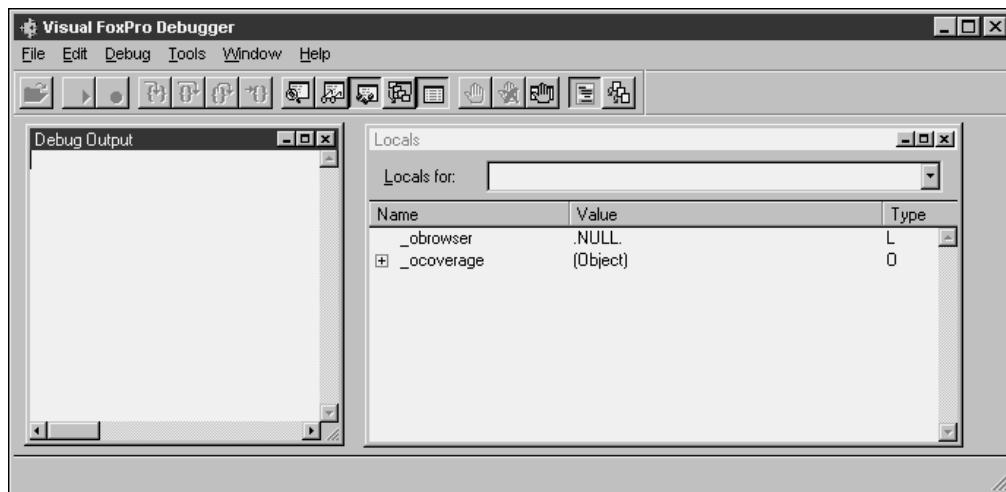
**Figure 21.1.** When you open the Coverage Profiler, you’ll be prompted for an existing log file.



**Figure 21.2.** Individual debug window menu options in the Tools menu mean that the Debugger is running in the FoxPro Frame.



**Figure 21.3.** Switching the Debugger to its own frame.



**Figure 21.4.** The Debugger running in its own frame.

Once you launch the Debugger in a separate frame, you'll have access to the Debugger Tools menu. Select the Tools, Coverage Logging menu option or the Toggle Coverage Logging button in the Debugger toolbar to open the Coverage dialog as shown in **Figure 21.5**. You can enter your own log file name, select an existing log file by clicking the ellipsis button, and choose whether to append new information to existing information or to overwrite the log file from scratch.

### Step 2: Run your app

This might seem obvious, but now that you've turned on coverage logging, you'll need to start your app—and run the parts of it that you want to log.

### Step 3: Turn off code logging

So now that you've got your data on your application, how do you turn off code logging? The easiest way is to issue the following command in the Command window, and be done with it.

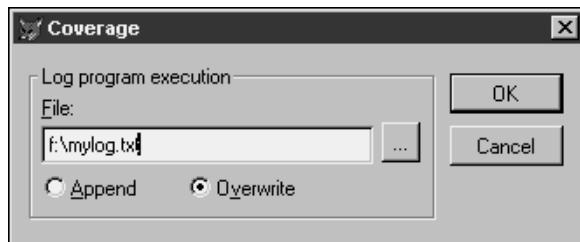
```
set coverage to
```

You can also quit VFP, and when you start up again, code logging will be stopped.

### Basic reports

If you take a look at your log file, say, in Windows Explorer, you'll find that it's a huge file. Just starting up a prototype of an application I've been working on, logging on, and shutting down creates a file of over 200K!

If you open it up, you'll see something like **Figure 21.6**.



**Figure 21.5.** Enter the name of the log file to be used by the Coverage Profiler.

A screenshot of a Windows Notepad window titled "mylog - Notepad". The window contains a large amount of text representing a coverage log. The text is in a fixed-width font and follows a specific format for each line, such as "15.817275,,it,85,f:\cct\w2\source\it.fxp,1". The content is too long to list here but represents the execution details of a VFP application.

**Figure 21.6.** A sample coverage log file.

Each line has the same format: a comma-delimited list of values:

- Execution time
- Class executing the code
- Object, method, or procedure in which the code is found or called
- Line number within the method or procedure
- Fully qualified file name
- Call stack level

Although the Coverage Profiler existed in VFP 5.0, the Call Stack Level value was added in VFP 6.0. And if you've not taken my advice and grabbed SP3 for Visual Studio, you'll miss a pretty nice enhancement that has to do with the Coverage Profiler. The precision of the

execution time value has been increased to six digits. This can be particularly handy when you're timing particularly fast commands or operations.

And there can be literally hundreds of thousands or millions of lines like this. The Coverage Profiler digests this information and allows you to create a number of reports that give you the information you want in a more easily understandable format.

### **Getting the Coverage Profiler to process a log file**

Now it's time to run the Coverage Profiler from the Tools menu in the Visual FoxPro window. You'll be asked to select a log file, and then the Coverage Profiler will do its thing. It will crank for a while (depending on how big the file is, of course), flash a few messages in the upper right corner through WAIT WINDOW commands, and then display the results in the Coverage Profiler window, as shown in **Figure 21.7**.

There's actually a lot of options here—I'll walk through them one by one.

### **Source List and Source Code panes**

Two windows appear when you open the Coverage Profiler. The top window, a two-column affair, is the Source List pane, and its purpose is to list discrete items that can be selected for analysis. For example, each class or program (shown as an “All Classes, Objects, Procs” item) is available in the left column. The source file for the class or program is displayed in the right column.

The bottom window is the Source Code pane, and, as you might have guessed, displays the source code for the particular item selected in the Source List pane. The display of the Source Code pane changes according to options you select; I'll get to this in a second.

### **The Coverage Profiler toolbar**

The toolbar on top of the Source List pane contains 10 buttons:

- Open—Opens a new log file for profiling.
- Save—Saves the coverage results to a table of your choosing.
- Statistics—Opens the Statistics dialog.
- Add-Ins—Adds an Add-In to the Coverage Profiler.
- Options—Opens the Coverage Profiler Options dialog.
- Coverage Mode—Toggles the Source Code pane to Coverage Mode.
- Profile Mode—Toggles the Source Code pane to Profile Mode.
- Preview Mode—Toggles the Source Code pane to Preview Mode.
- Zoom Mode—Toggles the Source Code pane to Zoom Mode.
- Find—Opens the Find dialog.

These are each covered in more depth shortly.

### **The Coverage Profiler context (shortcut) menu**

Right-clicking any component of the Coverage Profiler (the toolbar or either of the Source panes) brings forward the Coverage Profiler context menu, as shown in **Figure 21.8**.

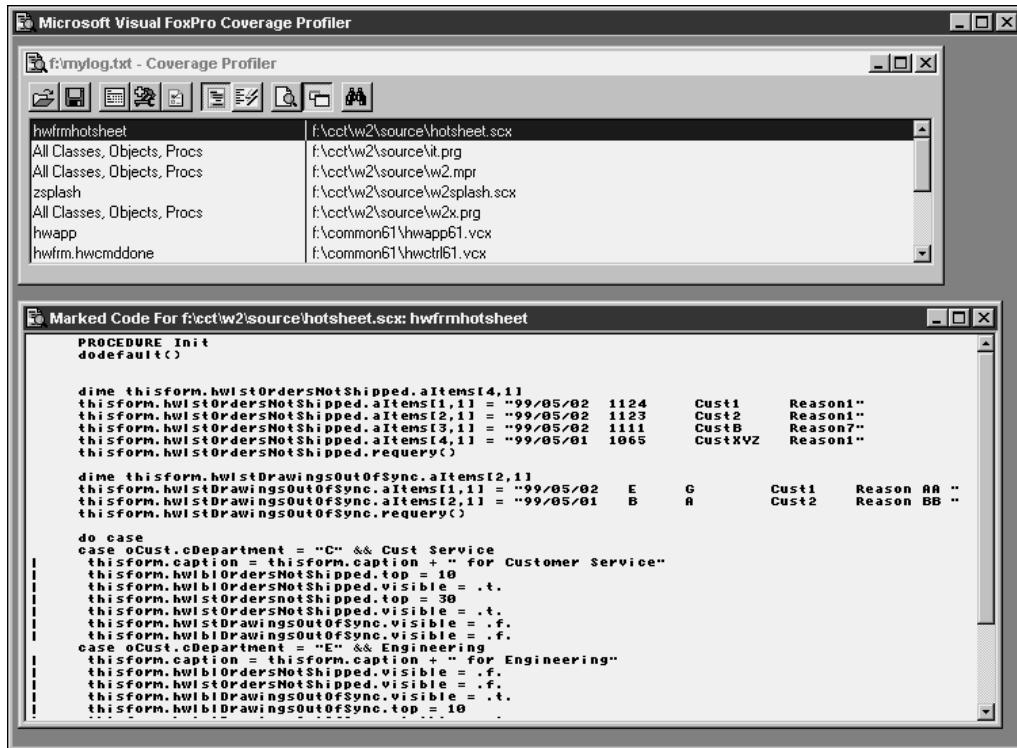


Figure 21.7. The Coverage Profiler window has two panes.

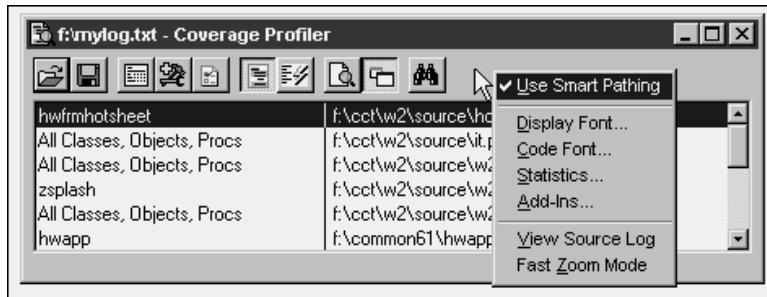


Figure 21.8. The Coverage Profiler context menu.

This menu has the following options:

- Use Smart Pathing—By default, the Coverage Profiler will prompt you to locate source files referenced in the coverage log. You can have the Coverage Profiler assume that folders that you've selected contain additional source files that it is

looking for. If you have multiple files with the same name in different folders, you might not want to check this option.

- Cascade Windows—If you are running the Coverage Profiler in its own frame, clicking this option will cascade the Source List and Source Code panes, and any other windows that might be open. Note that this menu option doesn't appear in Figure 21.8, because I was not running the Coverage Profiler in its own frame for this shot.
- Display Font—Allows you to choose the font for the Source List pane.
- Code Font—Allows you to choose the font for the Source Code pane.
- Statistics—Opens the Statistics dialog.
- Add-Ins—Allows you to add an Add-In.
- View Source Log—Opens the raw log file.
- Fast Zoom Mode—Displays source code of marked files as soon as you click the file in the Source List pane. When you select an unmarked file, you must first double-click the file in the Source List pane to view the associated source code in the Source Code pane. You must be in Zoom mode or this menu option will be disabled.

### **Coverage Profiler options**

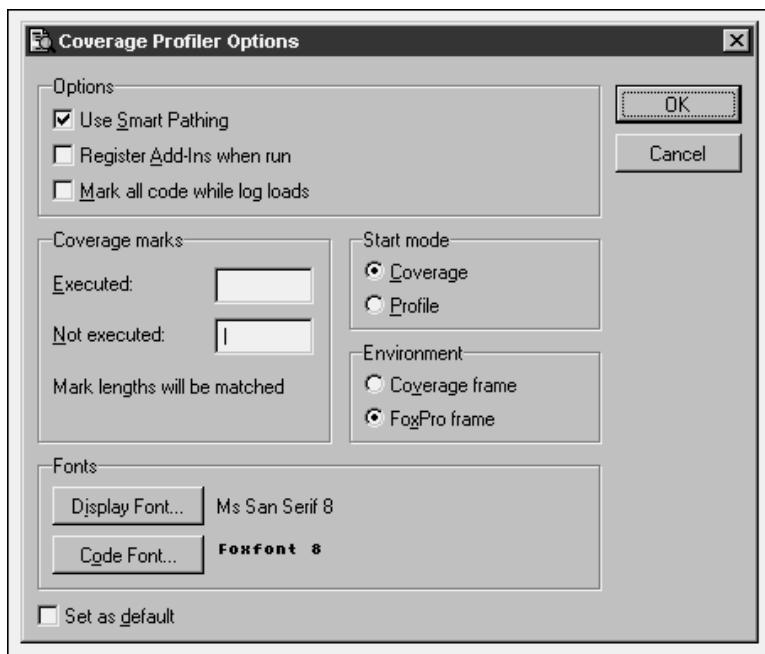
You can configure how the Coverage Profile will operate by setting options in the Coverage Profiler Options dialog, as shown in **Figure 21.9**.

#### **Options**

Checking the Use Smart Pathing check box will perform the same function as selecting the Use Smart Pathing menu option in the Coverage Profiler context menu. Checking the Register Add-Ins when run check box will automatically register an Add-In with the Coverage Profiler the first time you run it. The Register this Add-In after running check box in the Coverage Profiler Add-Ins dialog will automatically be checked. Checking the Mark all code while log loads check box will mark each line of code as appropriate while the Coverage Profiler processes the log file. If you have a lot of large files, this can slow down the initial load of the Coverage Profiler. On the other hand, once it's done, it's done!

#### **Coverage marks**

You can choose which characters appear to the left of each line of executed or not executed code in the Source Code pane. By default, every line that was not executed is marked with a vertical bar (or pipe). However, some lines, such as comments, are not marked as executed, and thus it can be difficult to scan through a listing and easily distinguish between executed lines and simple comments. You can use different length strings, such as "E >>>" for Executed and "|" for not executed—two spaces will be added to the single character pipe so that the display of the code remains consistent.



**Figure 21.9.** The Coverage Profiler Options dialog.

### **Fonts**

You can choose the fonts you want to display in the Source List pane and Source Code pane—these buttons work the same as their counterparts in the Coverage Profiler's context menu.

### **Start mode**

You can choose for the Coverage Profiler to display the Source Code pane in either Coverage mode or Profile mode. You might want to consider using Profile mode, because you can get all of the information in Coverage mode (code that wasn't executed is listed with “0 hits”) plus additional information—the time required for execution.

### **Environment**

You can choose for the Coverage Profiler to display in its own window, separate from the Visual FoxPro window (Coverage frame), or within the Visual FoxPro environment (FoxPro frame). This is just like the choice you have with the Debugger.

### **Set as default**

This is yet another example of an inconsistent (and dopey) user interface. There are about five different ways to save the settings you select in one dialog or another as your default settings—and no two ways are the same. If you want the changes you make in the Coverage Profiler Options dialog to be saved, check this check box and then click the OK button. If you make

changes in the Options dialog and click OK without checking this check box, the settings will persist just for the current session of the Coverage Profiler, not the current Visual FoxPro session. The Coverage Profiler Options settings are saved in the Windows Registry.

### **Coverage mode vs. Profile mode**

You can select whether you want to display the results of the Profiler (the contents of the Source Code pane) in Coverage mode or Profile mode.

#### **Coverage mode**

Coverage means that you'll see which lines of code were executed and which were not. Figure 21.7 shows you the display of a class in Coverage mode.

#### **Profile mode**

Profile mode allows you to determine how long each line of code took to execute. **Figure 21.10** shows you the same class in Profile mode.

| Line | Hits   | Count | Avg      | Code Snippet                                                                             |
|------|--------|-------|----------|------------------------------------------------------------------------------------------|
| 1    | 1 Hit  | 1st   | 0.000087 | Avg 0.000087 dodefault()                                                                 |
| 2    | 1 Hit  | 1st   | 0.000078 | Avg 0.000078 PROCEDURE INIT                                                              |
| 3    | 1 Hit  | 1st   | 0.000075 | Avg 0.000075 thisform.hwlordersNotShipped.items[4,1]                                     |
| 4    | 1 Hit  | 1st   | 0.000072 | Avg 0.000072 thisform.hwlordersNotShipped.items[1,1] = "99/05/02 1124 Cust1 Reason1"     |
| 5    | 1 Hit  | 1st   | 0.000072 | Avg 0.000072 thisform.hwlordersNotShipped.items[2,1] = "99/05/02 1123 Cust2 Reason2"     |
| 6    | 1 Hit  | 1st   | 0.000072 | Avg 0.000072 thisform.hwlordersNotShipped.items[3,1] = "99/05/02 1121 Cust3 Reason3"     |
| 7    | 1 Hit  | 1st   | 0.000072 | Avg 0.000072 thisform.hwlordersNotShipped.items[4,1] = "99/05/01 1065 CustXVZ ReasonXVZ" |
| 8    | 1 Hit  | 1st   | 0.000249 | Avg 0.000249 thisform.hwlordersNotShipped.requery()                                      |
| 9    | 1 Hit  | 1st   | 0.000103 | Avg 0.000103 dime thisform.hwlDrawingsOutSync.items[2,1]                                 |
| 10   | 1 Hit  | 1st   | 0.000099 | Avg 0.000099 thisform.hwlDrawingsOutSync.items[1,1] = "99/05/02 E G Cust1 Reason BB "    |
| 11   | 1 Hit  | 1st   | 0.000072 | Avg 0.000072 thisform.hwlDrawingsOutSync.items[2,1] = "99/05/01 B R Cust2 Reason BB "    |
| 12   | 1 Hit  | 1st   | 0.000206 | Avg 0.000206 thisform.hwlDrawingsOutSync.requery()                                       |
| 13   | 1 Hit  | 1st   | 0.000000 | Avg 0.000000 do case                                                                     |
| 14   | 1 Hit  | 1st   | 0.000000 | Avg 0.000000 case oCUST.cBDepartment = "C" & Cust Service                                |
| 15   | 1 Hit  | 1st   | 0.000000 | Avg 0.000000 thisform.caption = thisform.caption + " for Customer Service"               |
| 16   | 1 Hit  | 1st   | 0.000000 | Avg 0.000000 thisform.hwlOrdersNotShipped.top = 10                                       |
| 17   | 0 Hits | 1st   | 0.000000 | Avg 0.000000 thisform.hwlOrdersNotShipped.top = 30                                       |
| 18   | 1 Hit  | 1st   | 0.000000 | Avg 0.000000 thisform.hwlOrdersNotShipped.visible = .t.                                  |
| 19   | 1 Hit  | 1st   | 0.000000 | Avg 0.000000 thisform.hwlDrawingsOutSync.visible = .f.                                   |
| 20   | 0 Hits | 1st   | 0.000000 | Avg 0.000000 thisform.hwlDrawingsOutSync.visible = .t.                                   |
| 21   | 1 Hit  | 1st   | 0.000000 | Avg 0.000000 thisform.hwlOrdersNotShipped.requery()                                      |
| 22   | 1 Hit  | 1st   | 0.000000 | Avg 0.000000 thisform.caption = thisform.caption + " for Engineering"                    |
| 23   | 0 Hits | 1st   | 0.000000 | Avg 0.000000 thisform.hwlOrdersNotShipped.visible = .f.                                  |
| 24   | 1 Hit  | 1st   | 0.000000 | Avg 0.000000 thisform.hwlOrdersNotShipped.requery()                                      |
| 25   | 0 Hits | 1st   | 0.000000 | Avg 0.000000 thisform.hwlDrawingsOutSync.visible = .t.                                   |

**Figure 21.10.** Profile mode in the Coverage Profiler shows a listing of each line of code, how many times it was executed, and how long each execution took.

### **Zoom mode vs. Preview mode**

You can select whether you want to display the results of the Profiler (the contents of the Source Code pane) in Zoom mode or Preview mode.

#### **Zoom mode**

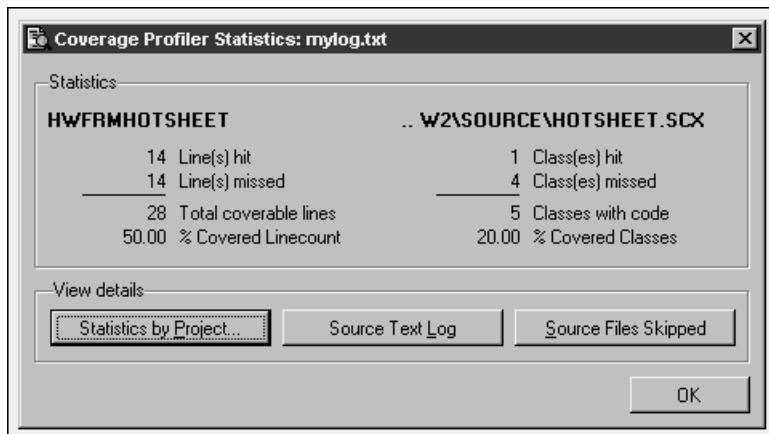
Zoom mode displays the Source Code pane in a separate window that can be resized independently of the Source List pane. You can also select Fast Zoom (described later) in Zoom mode.

#### **Preview mode**

Preview mode displays the Source Code pane in a window that is docked with the Source List pane. Fast Zoom is not available in Preview mode.

## Statistics

You don't necessarily want to go through every item in the Source List just to determine which lines of code haven't been run, do you? The Statistics dialog allows you to get a bird's eye view of the results of the Coverage Profiler's work. See **Figure 21.11**.



**Figure 21.11.** The Coverage Profiler Statistics dialog.

There are four pieces to the Statistics dialog. The first simply displays a summary of the number of lines of code, the classes the Coverage Profiler hit, and how many it missed. If you get 100% in both of these areas, you're probably in pretty good shape, but if you've got a lot of misses, you need to do more testing.

Clicking the Statistics by Project button will open a dialog asking you to pick the project file, and will then open a report that displays the contents of the project file, together with statistics on each file in the project. See **Figure 21.12**.

You can view the raw log file by clicking the Source Text Log button, or see a list of source files skipped (in a browse window) by clicking Source Files Skipped.

## Find

So far, you've seen how each tool works, but not much as far as real-life needs. The Find button—also new in Visual Studio SP3—is one of those features that you don't know you need until you need it. Think about two common situations that are likely to happen when you're using the Coverage Profiler. One is when you're looking for a chunk of code that you don't think has been executed. A second is when you're trying to determine why a particular module or function is running so slowly. In both cases, you might suspect a specific area of the code. With the Find button, you can look for a keyword or phrase in that area of the code and move right to it, instead of having to wade through possibly pages and pages of code to find the area of interest.

Clicking the Find button opens the Find dialog as shown in **Figure 21.13**.

```

Coverage Profiler Statistics by Project...
f:\cct\w2\source\w2.pjx

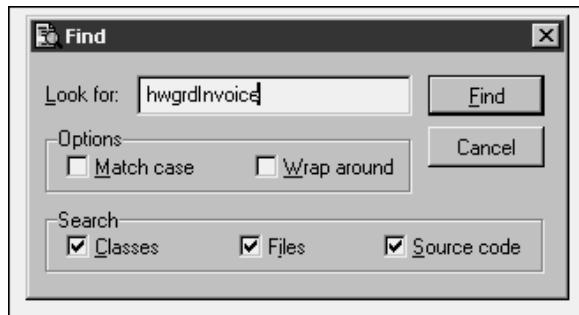
File skipped or not logged: f:\cct\w2\source\w2.vcx
f:\cct\w2\source\w2.mpr
    182 Line(s) hit
    188 Total coverable lines
    96.81 % Covered Linecount

f:\cct\w2\source\hotsheet.scx
    14 Line(s) hit
    28 Total coverable lines
    1 Class(es) hit
    5 Classes with code
    50.00 % Covered Linecount
    20.00 % Covered Classes

File skipped or not logged: f:\common61\zsysinfo.prg
File skipped or not logged: f:\common61\zuserma.scx
File skipped or not logged: f:\common61\zsysinfo.vcx
File skipped or not logged: f:\common61\zsysinfv.scx
File skipped or not logged: f:\cct\w2\source\sbcust.scx
f:\common61\hwapp61.vcx

```

**Figure 21.12.** A sample Coverage Profiler Statistics report.



**Figure 21.13.** The Find dialog in the Coverage Profiler allows you to zero in on a specific word or phrase.

You can, of course, choose to match case by selecting the Match case check box. If you start your search in the middle of a file, you can have the Find mechanism continue its search at the beginning of the file once it reaches the end by selecting the Wrap around check box.

You can choose which items to search in—any combination of classes, files, and source code.

## Open

Once you've loaded the Coverage Profiler, it would be a waste if you had to close it just to examine a different log file. This button allows you to close the current log file and open another one.

## Save

You might want to save the results of the Coverage Profiler to a table of your choosing. Clicking the button will display a dialog prompting you for a table name, and provide a default value for you. The structure of the table is shown in **Figure 21.14**.

The Marked and Profiled memo fields contain the text of the Source Code panes in Coverage and Profile modes. If you think about it, COVERAGE.APP is essentially a user interface for this table, much like the Form Designer is a user interface for an .SCX table.

| Hostfile                           | Objclass     | Marked | Profiled | Coverable | Covered | Objtotal | Objhits |
|------------------------------------|--------------|--------|----------|-----------|---------|----------|---------|
| f:\ccct\w2\source\hwfmhotsheet.scx | hwfmhotsheet | Memo   | memo     | 28        | 14      | 5        | 1       |
| f:\ccct\w2\source\wl.prg           |              | memo   | memo     | 0         | 0       | 0        | 0       |
| f:\ccct\w2\source\wl2.mpr          |              | memo   | memo     | 0         | 0       | 0        | 0       |
| f:\ccct\w2\source\wl2splash.scx    | zsplash      | memo   | memo     | 0         | 0       | 1        | 1       |
| f:\ccct\w2\source\wl2x.prg         |              | memo   | memo     | 0         | 0       | 0        | 0       |

**Figure 21.14.** You can save the results of the Coverage Profiler to a table with this structure.

## Customizing the Coverage Profiler with Add-Ins

An Add-In is a Visual FoxPro program that adds or changes the functionality of the Coverage Profiler. You can create your own Add-Ins to add additional controls, reports, modified user interfaces, and other enhancements to the Coverage Profiler.

Instead of repeating work that's been done elsewhere, however, I'll point you to a pair of great resources. First, check out Markus Egger's Coverage Snippet Analyzer, available at [www.eps-software.com](http://www.eps-software.com), for one example of an Add-In. Second, Lisa Slater's excellent article on subclassing the Coverage Engine tells you all you'll ever want to know about the subject. It's available at

<http://msdn.microsoft.com/vfoxpro/technical/articles/applications.asp>

You might want to grab the source code for the Coverage Profiler (as well as the source for other VFP tools) from XSOURCE.ZIP—found in the VFP98\TOOLS\XSOURCE directory.



# Chapter 22

## Using MSDN Help

**Back in the olden days, you learned a language by reading the reference guide and paging through the 20-page tutorial. Once you were past that, your sources of help were scarce: the index to the documentation, a magazine with a relevant article, or, if all else failed, a toll call to technical support. Nowadays, there's an abundance of resources for getting help past the product's documentation. Microsoft's help encompasses three resources: the online help that you can install with the product, and two more Web-based help resources. In this chapter, I'll discuss each of these and how to use them.**

Most people are familiar with pressing F1 and getting a screen of online help. Unfortunately, most people just stop there. There are a number of powerful features in the online help for Visual Studio, and using them is a necessity for a product that powerful and complex. There are also two large Web-based databases that provide support for the product, but many people aren't aware of them or don't know how to use them properly. It's time to explore.

### The MSDN Library

Microsoft has long had a resource called Microsoft Developer Network (MSDN). Although the makeup of this support program has changed over the years, its intent is still the same—to provide a comprehensive source of documentation and other technical resources for developers. The basic idea behind MSDN is a fee-based, subscription-oriented support service that provides regular updates of products and documentation during the life of the subscription.

With Visual Studio 6.0, the online help files for all products—Visual Basic, Visual C++, Visual FoxPro, Visual InterDev, Visual J++, and other ancillary products—can be accessed under one unified engine. This unified engine is called the MSDN Library, and is a subset of the MSDN product. As VS has become more and more integrated, this makes sense, because it would be both a waste and a source of errors or inconsistencies to duplicate topics that were relevant to more than one product.

The help files for each individual product have all been converted to HTML Help; thus each has its own .CHM (Compiled HTML Help) file that can be accessed on its own, just as any .CHM file can be. However, the MSDN Library engine provides a single access point to all of the files as well.

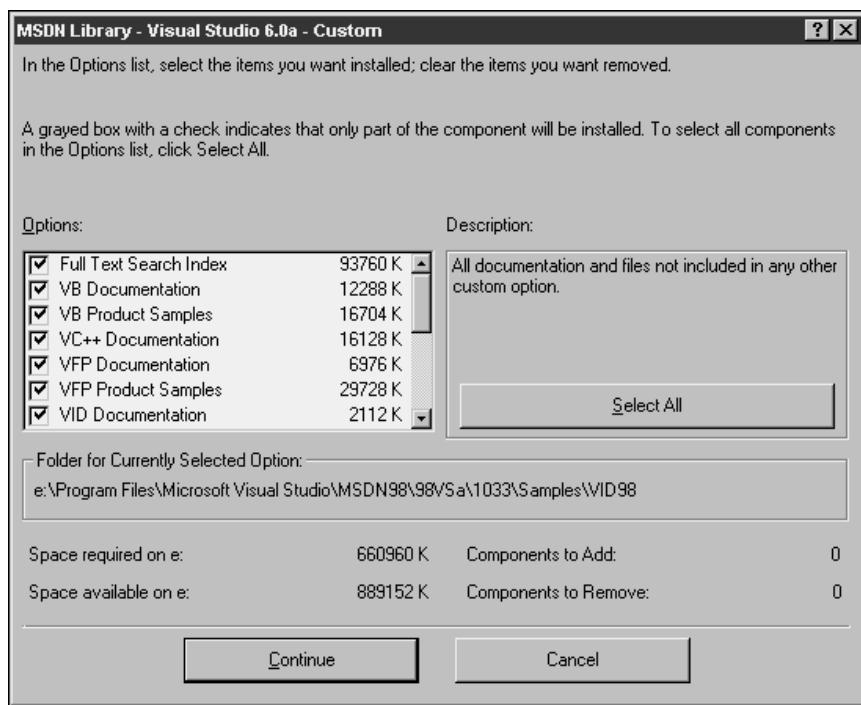
You don't have to install all components of the MSDN Library if you just want to access the help files for a particular product, and many people don't because of disk-space limitations or just a purist's sense of "I only want product X on my machine." If you go that route, however, you'll likely run into situations where a help topic you're trying to access hasn't been installed, and you'll be prompted for the CD. So I recommend you install the entire MSDN Library, even if you don't use all of the products.

## Installation

In order to take advantage of the MSDN Library, you first have to install it properly. The MSDN Library comes on its own CDs (yes, that's plural), and while it can be installed as part of the process of installing Visual Studio, it can also be installed separately. And because many Fox developers probably only installed the Visual FoxPro part of MSDN, it's a good idea to review the independent install process.

First, dig out your MSDN CDs—either as part of the Visual Studio CDs or, if you subscribe to the MSDN support service, the most recent MSDN Library CDs you've got. As of this writing, there are two CDs. Put the first one ("Disk 1") in your CD-ROM drive and wait for the auto-installer to kick in. If it doesn't, run the SETUP.EXE program in the root of CD 1.

After some housekeeping tasks such as checking disk space, you'll be presented with a screen that looks something like **Figure 22.1**.



**Figure 22.1.** Installing or reinstalling MSDN Help is done from this screen.

If you select all the check boxes, you'll end up with about 700 MB of materials. If you don't have enough space on your C drive to keep all of this, here's a hint: Although the install program leads you to install all files in the same Program Files\Microsoft Visual Studio directory, in which all programs themselves are placed, you don't have to do so. I keep MSDN Help on a different drive that has more room than that dinky 2 GB partition on C. In fact, I

believe you can install all of the help files on a network drive. (I haven't done this myself, but I don't see why it wouldn't work.)

## Opening MSDN Help

Once you've got it all installed, Visual FoxPro should be pointing to the MSDN Help engine, not to the VFP .CHM file itself. You can tell by looking in the File Locations tab in the Tools, Options dialog. In the Help File entry, you should see something like this:

```
C:\Program Files\Microsoft Visual Studio\MSDN98\98VSA\1033\MSDNVS6A.COL
```

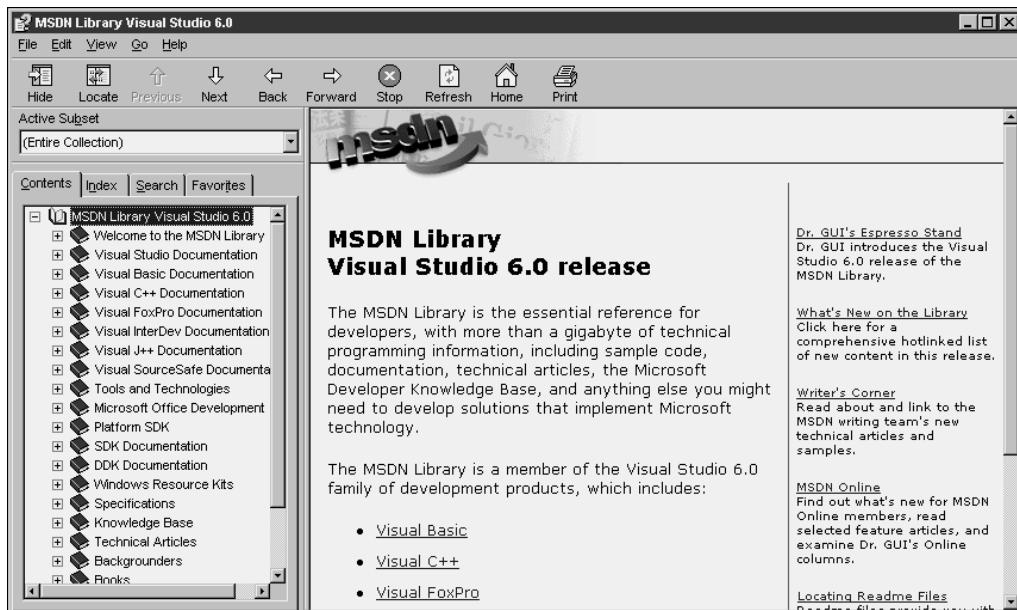
To access help while in VFP, just select the Help, Microsoft Visual FoxPro Help menu option, and the main MSDN Library screen will be displayed, as shown in **Figure 22.2**.

You can also call up help by pressing F1. This brings up context-sensitive help—if you've got a particular dialog open, such as the Form Designer, the help topic for that dialog will be displayed automatically.

You can also find help on the commands and functions by using the Command window. You simply issue the Help command followed by the keyword you're interested in; for example:

```
help set strictdate
```

will bring forward the help topic on the SET STRICTDATE command.



**Figure 22.2.** The main MSDN Library screen.

## Features of MSDN Help

Once you've opened the Help window, a number of features are available to you. First off, the basic engine is the HTML Help application, so you get all standard features of HTML Help.

The left pane has a four-tab page frame, with each tab providing a different set of features to access one or more entries. Once you select an entry, the help topic will be displayed in the right pane.

### The Contents tab

The Contents tab displays a TreeView control with a multi-level hierarchy of entries. As with other Explorer interfaces, you can drill down through the hierarchy by clicking on entries that have a (+) in front of them, or close up a level of entries by clicking on the (-) in front of the top-level entry.

If you examine Figure 22.2 carefully, you'll see that the MSDN Library contains much more than just reference material. The Knowledge Base, for example, contains more than 40,000 articles about products, bug reports, bug fixes and workarounds, documentation errors, and answers to frequently asked technical support questions.

There are also numerous technical articles, backgrounders, excerpts from books and conference papers, and even the full text of about a dozen Microsoft Press books covering a variety of Windows development topics.

### The Index tab

The Index tab displays a list of every keyword or phrase in the entire index of the MSDN Library. This list is organized alphabetically, and some entries have subentries. You can often view topics for both the entry and its subentries. However, not all entries have associated topics. For example, if you click the “~ (tilde)” entry, you'll get topics describing Unary Arithmetic Operators. You can also click one of the subentries, such as “specifying destructors.” However, if you select the “! (exclamation point)” entry, you'll get a message indicating you have to select one of the subentries as shown in **Figure 22.3**.

### The Search tab

The Search tab has all sorts of slick capabilities. You can enter a word or expression in the combo box, and then click the List Topics button to see all topics where that word or expression was found. The number of matches is displayed midway below the List Topics and Display command buttons. In **Figure 22.4**, I've searched for the expression “config.fpw” and found 154 matches.

Because there could potentially be a lot of entries, the results are ranked by a super-secret algorithm known only by two programmers who are never allowed to ride in the same airplane together. If you examine the top few topics, you can noodle a couple of general rules. For instance, topics with the keyword or expression in the title of the help topic are ranked higher than other topics. Other factors that help determine the rank include the number of times the expression is found in the topic and how close the expression is to the beginning of the topic.

Once you have found a likely topic, click the Display button (or just double-click the selected topic in the list). The topic will be displayed in the right pane.

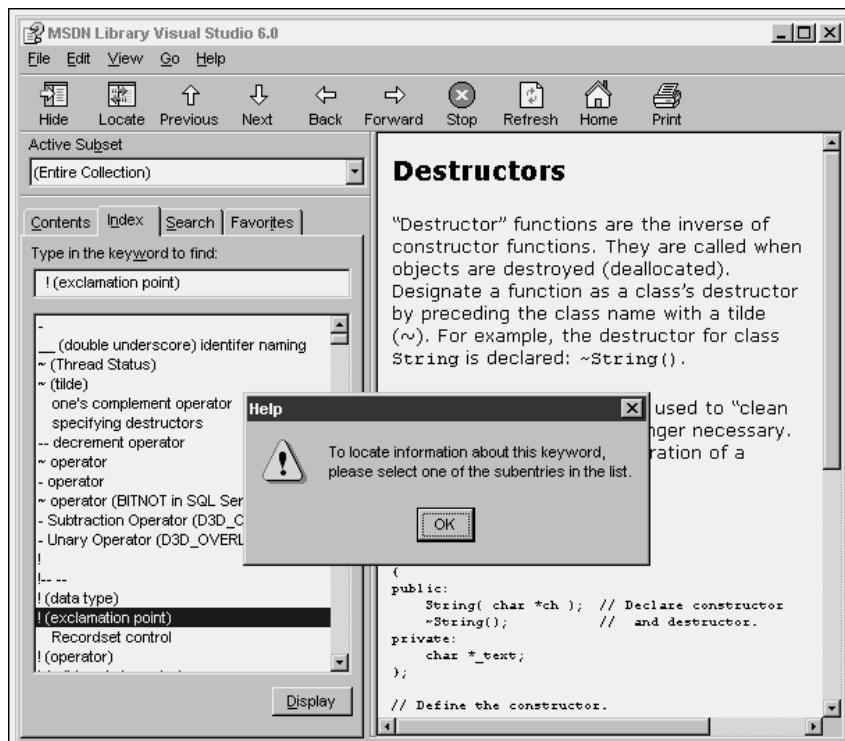
If you want to search for topics that contain an entire phrase, enclose the phrase in quotes. Otherwise, you'll get every topic that contains all of the words in the expression, but those

words can be anywhere in the topic—they don't have to be together in the order you specified. For example, in **Figure 22.5**, I've searched on the phrase "Upsizing to SQL Server" (without the quotes). The results included 148 topics, and as you see, all of the words were found—just not in proximity to each other.

You can match similar words (for example, if you search on "print" but want to see topics with "printing" and "printed" as well) by checking the Match similar words check box. Be careful, because this will find a large number of topics if you are searching for common words.

You can create Boolean expressions as well, using common Boolean operators such as AND and OR. You can click the button to the right of the combo box to select an allowable operator (AND, OR, NEAR, and NOT). It's tricky to use it—here's how:

1. Type the word "Strictdate" in the combo box.
2. Click the right-arrow button to display a context menu of the four operators.
3. Select the desired operator, such as OR. The context menu will disappear, and the operator will appear to the right of "Strictdate" in the combo box.



**Figure 22.3.** Not all entries in the MSDN Library Index have topics associated with them.

4. Type a second keyword (or an expression, surrounded by quotes) after the operator, so that you see

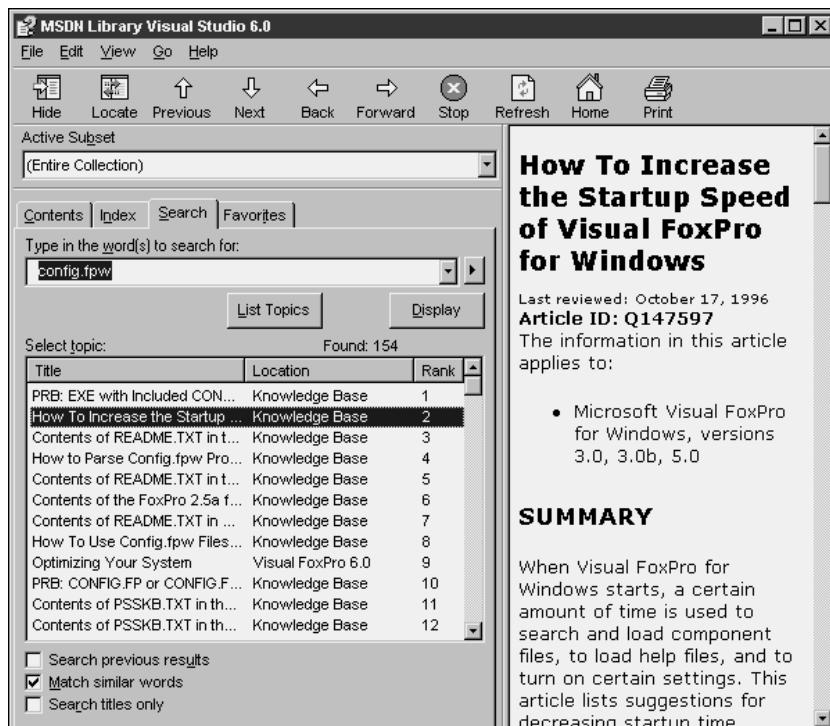
```
strictdate OR "Set Century"
```

You can choose to search only topic titles by checking the Search titles only check box. And you can narrow your search to just those topics in a previous results set by checking the Search previous results check box. For example, if you searched on “date” and got 487 matches, you can then search on “century”, and by checking the Search previous results check box, the “century” search will be performed just on those 487 matches, not every topic in the MSDN Library.

If you’re thinking it’s easier to just type the word OR, you’re right—once you know which operators are allowed.

### ***Whittling down the topics***

If you aren’t judicious about the expressions you use, you could end up with lots of topics. In fact, if you search on a word like “Print” or “Integer”, you’ll get so many that the search engine will stop after it finds 500 matches. (Did you wonder why there were so many keywords that had exactly 500 topics?)



**Figure 22.4.** Searching for the expression “config.fpw” found 154 matches.

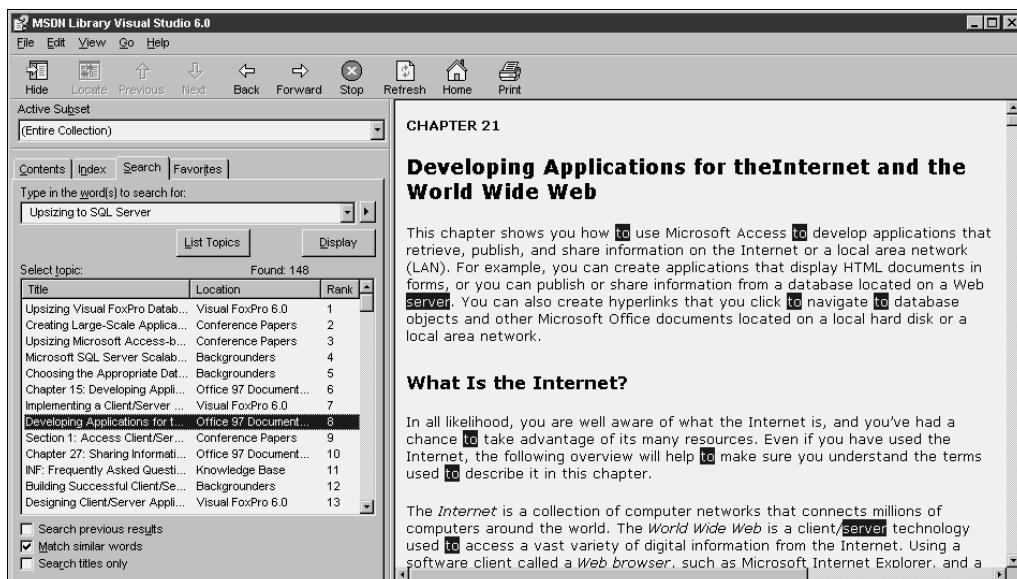
How can you narrow the list? First of all, use more words rather than fewer, and be sure to use quotes to search for the exact phrase. Uncheck the Match similar words check box, and, optionally, check the Search titles only. Once you've got a reasonable list, you can further refine your search by using the Search previous results check box.

### The Favorites tab

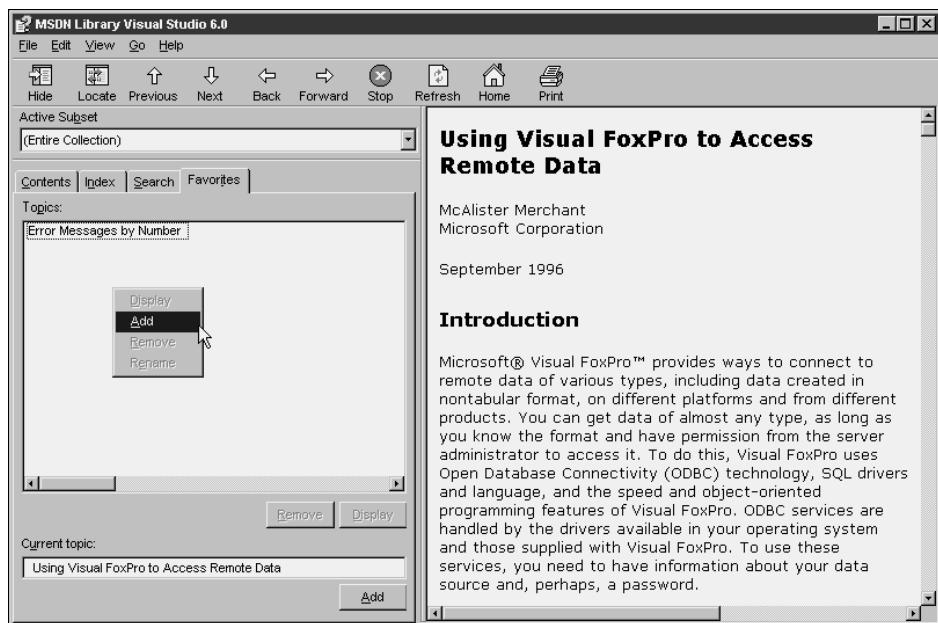
The last tab in the left pane is for Favorites. The use of Favorites is a bit counter-intuitive, I think. Instead of selecting the Favorites tab and then searching for a topic to add, you first have to find a topic and decide that you want to add it to Favorites.

To do so, once the topic is displayed in the right pane, switch to the Favorites tab. Then right-click in the list box, as shown in **Figure 22.6**. Select the Add menu option, and the topic displayed will be added to your list of favorites.

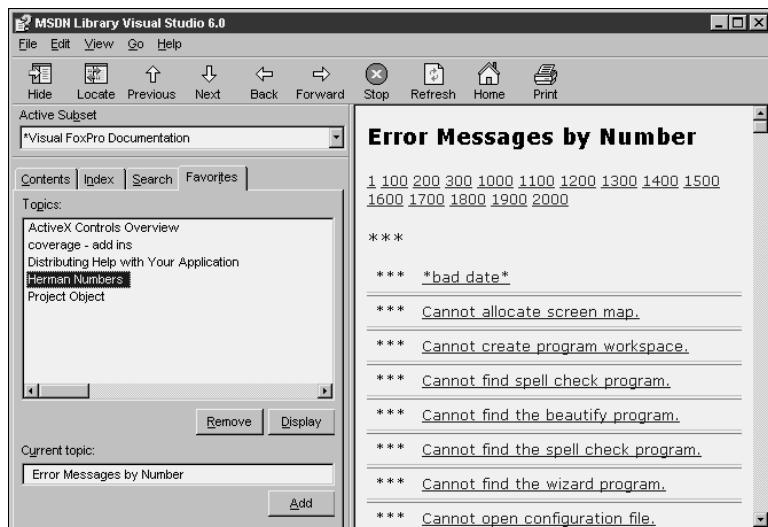
To remove a topic from the list, select the entry in the list box, right-click, and select the Remove menu option. You can also rename a topic entry if the entry that's provided isn't as intuitive as you would like for your own reference. Note that you're only renaming the topic in Favorites, not in all of the MSDN Library. In **Figure 22.7**, I've renamed the Error Messages by Number topic to "Herman Numbers" but you'll see that the topic still retains the original title, and if you look under the Visual FoxPro Reference help topics, "Error Messages by Number" is still there where it was originally.



**Figure 22.5.** Searching for the phrase "Upsizing to SQL Server" found 148 matches that had all four words somewhere in the topic.



**Figure 22.6.** Right-click in the Favorites tab list box once you've got a topic displayed in the right pane.



**Figure 22.7.** The Favorites topic has been renamed to "Herman Numbers" but the actual topic name is still shown in the Current topic text box in the lower left corner of the dialog.

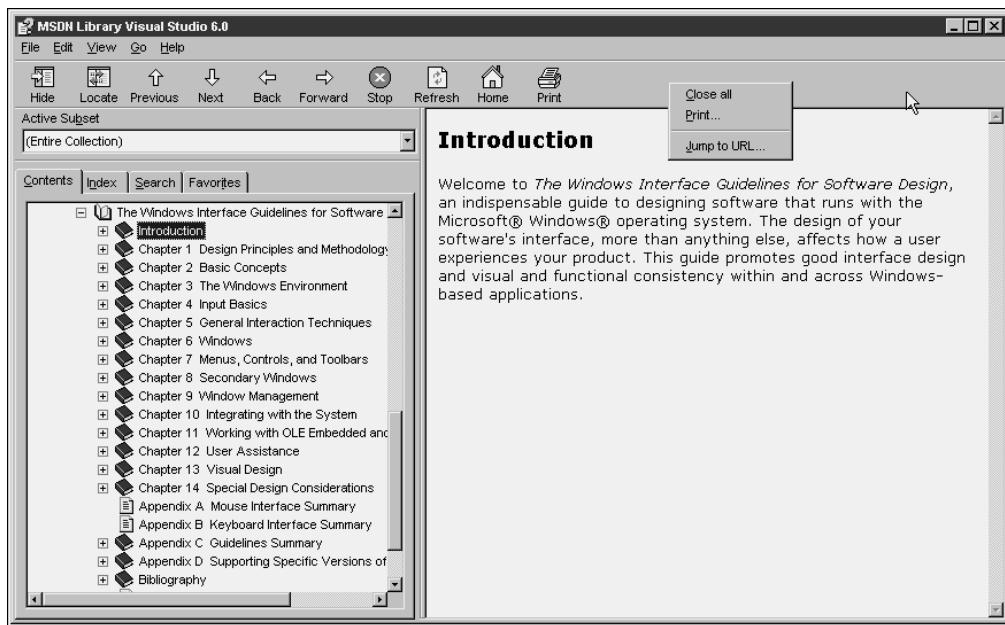
## The MSDN Library menu bar

The MSDN Library help engine is a program that runs on top of the HTML Help engine. That's why you see a menu bar at the top of the MSDN Library window, unlike regular HTML Help windows in other applications. I'm not talking about the browser-style buttons like Hide, Locate, Previous, Next, and so on—they come standard with HTML Help, and are pretty obvious in terms of what they do. However, a feature that might not be obvious is that you can right-click in the area next to the browser buttons and get a small context menu as shown in **Figure 22.8**.

Close All and Print do what they say—Close All shrinks the outline control in the Contents list box to the top-level entry, while Print displays the Print dialog in **Figure 22.9**, which is fairly self-explanatory. You can also access the Print Topics dialog from the File, Print menu option.

The Jump to URL menu option displays the dialog shown in **Figure 22.10**. You can display a Web page from the World Wide Web in the right pane as well as topics from the MSDN Library. This dialog shows you the URL of the current Web page and also allows you to enter a new URL. Note that simply entering [www.yourwebsite.com](http://www.yourwebsite.com) won't be enough—you need to prefix the address with `http://` or another mechanism as appropriate.

The MSDN Library menu bar also contains File, Edit, View, Go and Help menu pads, which I'll explain in the following sections.



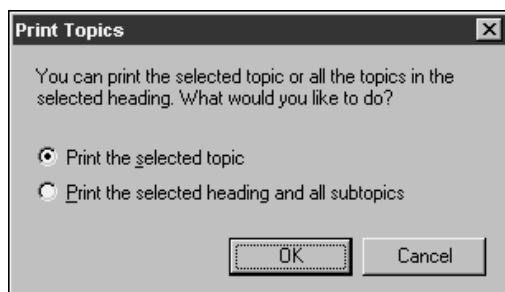
**Figure 22.8.** Right-clicking in the area next to the browser buttons displays a handy little context menu.

## File

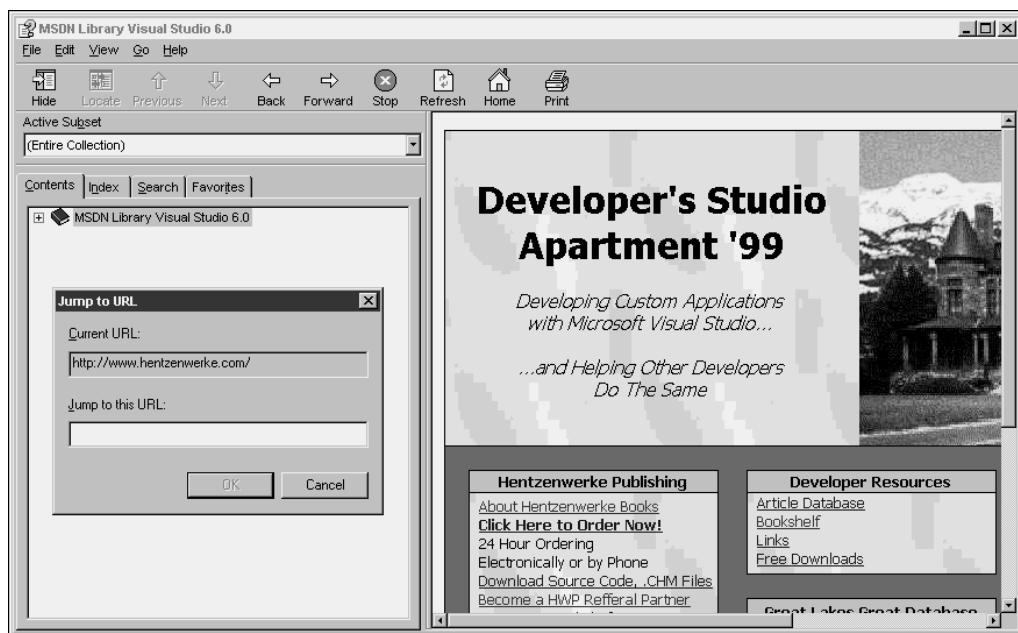
The File menu pad has just two options: Print and Exit. Selecting Print displays the dialog shown in Figure 22.9.

## Edit

The Edit menu pad has a limited number of choices—Copy, Select All, and Find in this Topic—that are all fairly self-explanatory. Clicking Find in this Topic displays the dialog in Figure 22.11.



**Figure 22.9.** You can print either a single topic or the topic and all subtopics through the File, Print menu option.



**Figure 22.10.** The Jump to URL dialog shows you the URL of the current Web page and also allows you to jump to a new one.

## View

The View menu pad, on the other hand, has a whole raft of options, as shown in **Figure 22.12**.

Locate Topic in Contents allows you to drill into the TreeView control and find where a displayed topic is displayed. For example, suppose you've added a topic to Favorites, but after a while, you realize you forgot where you got it, and you can't find it in the contents simply by trial and error. You can display the topic in the right pane, and then select this menu option to display where in the TreeView the topic is located. You can also do this by clicking the Locate button in the browser button bar.

The Navigation Tabs menu option is either checked or unchecked—it performs the same function as clicking the Show/Hide buttons between the menu and the two panes. The Contents, Index, Search, and Favorites menu options all allow you to bring forward a specific tab in the left pane. I can't see why you'd want to use the mouse to select one of these menu options, but their existence means you've also got keyboard shortcuts for those tabs.

The Stop menu option simply allows you to stop a search that's taking a long time or, if you're viewing content from the Web in the topic pane, to interrupt a download. Refresh allows you to refresh the content from the Web in the topic pane.

The Source menu option is enabled when a topic in the right pane has focus, and simply allows you to view the HTML underlying the topic.

When you do a search on a word or phrase, that word or phrase is highlighted in reverse video throughout the topic. This can make it very hard to read if there are a number of matches in the topic. You can uncheck the Highlights menu option to remove the reverse video highlighting in the current topic.

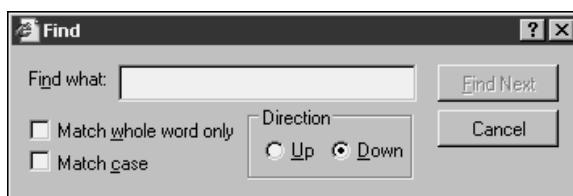
You can choose the size of the font you want topics displayed in the right pane by selecting the Fonts menu option. Note that this affects only the right pane—the left pane remains the same regardless of the font size selected.

The Internet Options menu option brings forward the same Internet Options dialog (with tabs for General, Security, Content, Connections, Programs, and Advanced) that you can get through the Tools, Internet Options menu option in Internet Explorer.

The last menu option is Subsets—I'll discuss this in more depth in the next section.

## Go

The Go menu pad contains menu options that allow you navigate through topics in the same manner that the browser buttons do. See **Figure 22.13**.



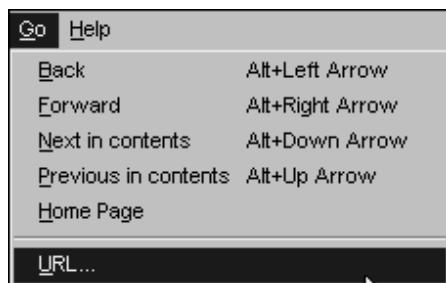
**Figure 22.11.** You can perform a variety of searches in a particular topic by clicking *Edit, Find* in this Topic.

## Help

In addition to help about using the MSDN Library and a useless About box (it only tells you the version of the HTML Help control, which, given the frequent upgrades of HTML Help, only serves to tell you that this version of the control is badly out of date), the Help menu pad has a great hidden feature—a direct link to the MSDN Web site's home page. Check it out!



**Figure 22.12.** The View drop-down menu has a large number of options.



**Figure 22.13.** The URL menu option opens the same Jump to URL dialog shown in Figure 22.10.

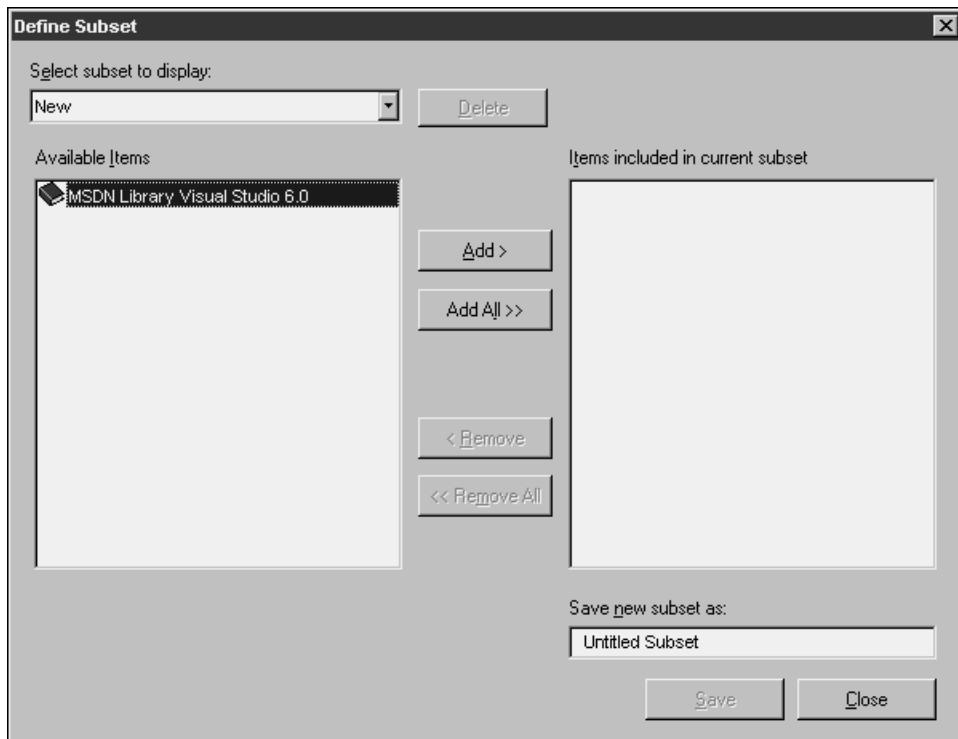
## Subsets

You can't help but have noticed by now that there is an awful lot of material in the MSDN Library. If you printed this all out on 8.5 x 11 paper, single-spaced with half-inch margins, you'd end up with about 30 four-drawer file cabinets full of paper. And it's likely that you're going to want to regularly use only part of the materials in the Library. You can winnow out materials that you normally don't want to use by creating subsets of topics that are of particular interest to you.

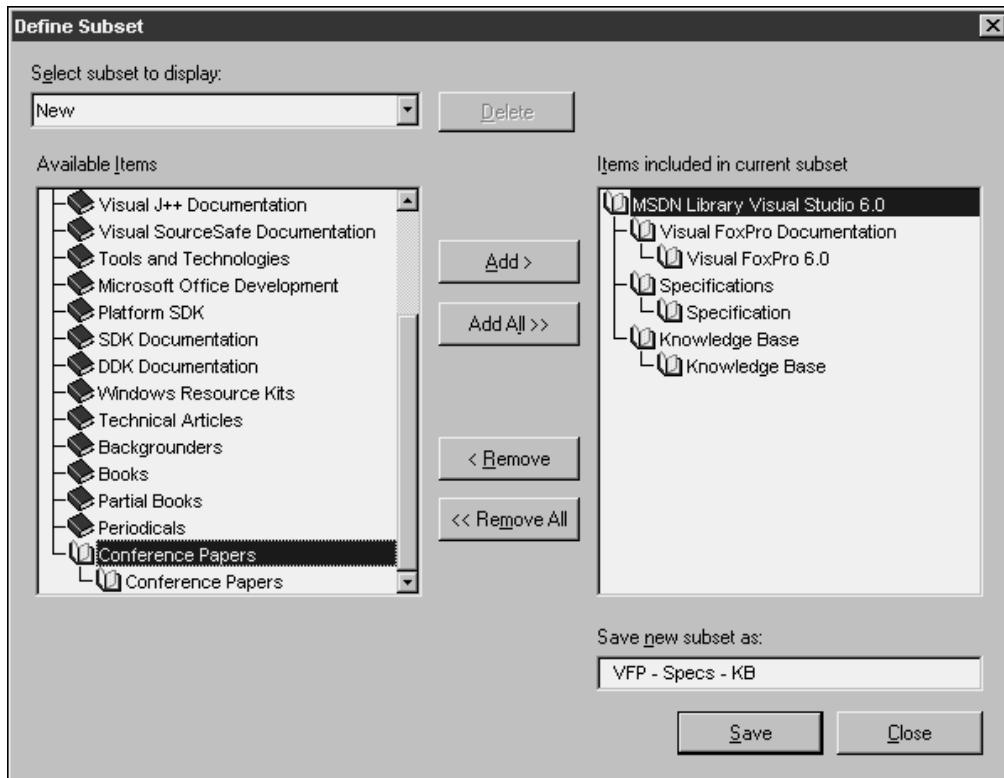
To create a subset, select the View, Define Subset menu option. The Define Subset dialog displays as shown in **Figure 22.14**.

The initial display of the Define Subset dialog isn't exactly intuitive—you need to double-click on the MSDN Library Visual Studio 6.0 entry to expand it into an outline of all available items. Once you do so, you can highlight an item and then click the Add button to move it to the subset you're defining, as shown in **Figure 22.15**.

Once you've added to the subset all the items you're interested in, enter a name in the Save new subset as text box in the lower right corner, and click the Save button.



**Figure 22.14.** You need to double-click on the MSDN Library Visual Studio 6.0 entry in the Available Items list box in order to view entries that can be moved into a subset.



**Figure 22.15.** Highlight an item and then click the Add button to include it in the current subset.

After you've created a subset, you'll see that the current subset is still displayed. How do you create yet another subset? All you need to do is select the New item in the Select subset to display combo box, and the two list boxes—Available Items and Items included in current subset—will be restored to their original state.

So now that you've defined a few subsets, how do you use them? Open the MSDN Library help window and select the Active Subset combo box. The subsets you've added will appear at the bottom of the list, as shown in **Figure 22.16**.

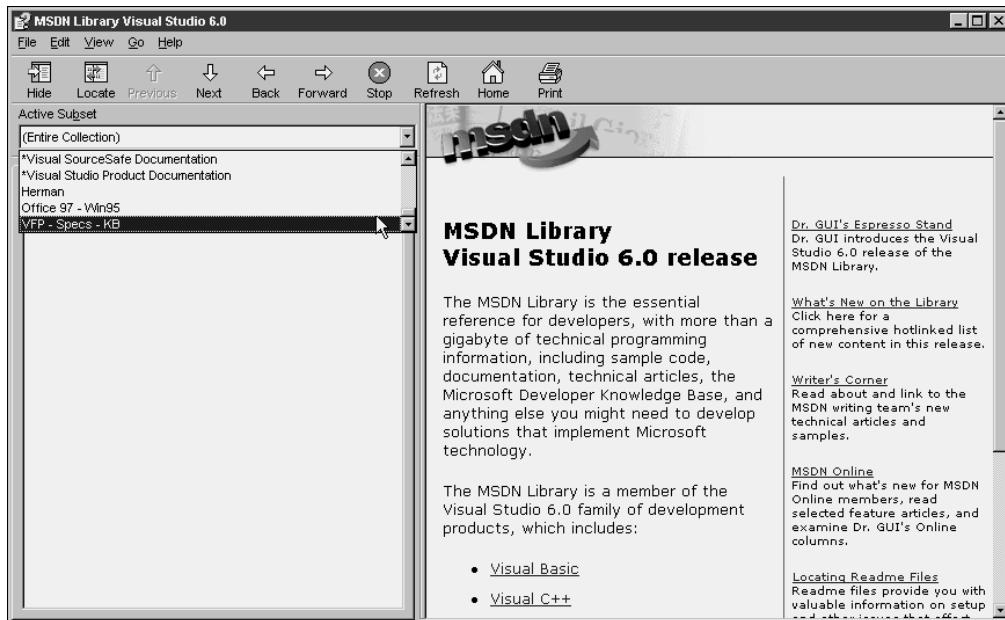
If you decide you don't need a subset anymore, you can remove it by selecting the View, Define Subset menu option, selecting the subset to rid in the Select subset to display combo box, and then clicking the Delete button to the right of the combo box. You will be prompted to verify your action.

## Help on the Web

If you've just bought a copy of Visual Studio, you'll get a copy of the MSDN Library as it stood at some specific date in time—perhaps three months before the release of the product, given the lead time required to format and proof articles, press CDs, and package everything in

boxes. Then, that copy of Visual Studio might have sat upon a shelf for another indeterminate length of time, resulting in a help file that could be anywhere from three months to over a year out of date.

In the brick-making business, that might not amount to much, since state-of-the-art 4000 years ago is still pretty much state-of-the-art now. But in the software biz, well, it's difficult to keep current no matter how much time you devote to it. Having your help files on a static medium such as a CD is one sure way to ensure you stay out of touch with the newest news.



**Figure 22.16.** The subsets you create yourself are found at the bottom of the contents of the Active Subset combo box.

## MSDN

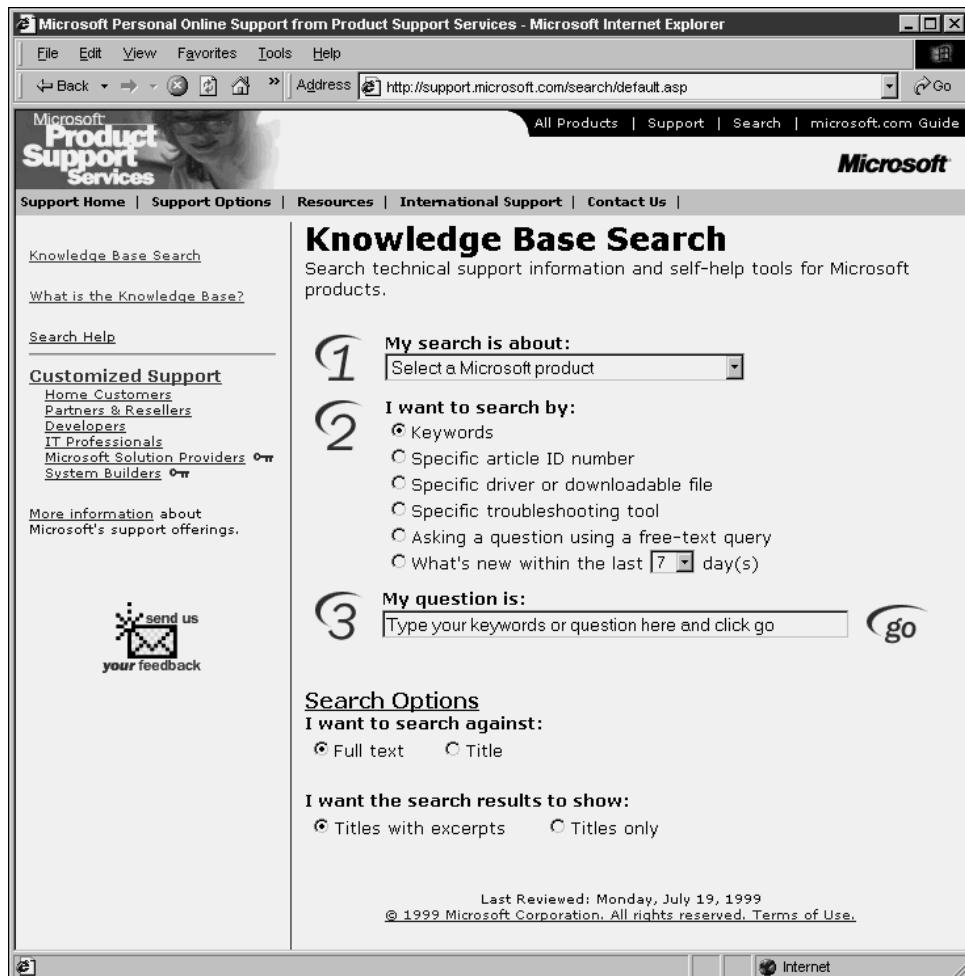
You can access the very latest version of MSDN on Microsoft's Web site at <http://msdn.microsoft.com>.

You can get to this site in a number of ways. First, you can just type the URL in your browser, but, well, blech! You can get there directly via the Help menu in the MSDN Library Help window, or through Visual FoxPro via the Help, Microsoft on the Web, MSDN Online menu option. I heartily encourage you to pop up there soon and scout around. You'll have access to the very latest articles, bug fixes, and other product information.

## The Microsoft Knowledge Base

There's another database on Microsoft's Web site that's just as good—the Knowledge Base, as shown in **Figure 22.17**. The URL for this tool is <http://support.microsoft.com>.

I won't spend any time describing how to use this site; it's fairly straightforward, and the interface is updated regularly. Nonetheless, it's another site you'll want to keep bookmarked or memorized for regular access.



**Figure 22.17.** Use the Microsoft Knowledge Base to get information about any Microsoft product.

# Section V

## Advanced Development Tools and Techniques

My wife has a recipe for brownies that is just out of this world. However, there are a lot of different ways to make brownies, and after a block party where someone else had brought a home-baked cake with a rich frosting, my wife had an idea for a new recipe. She tweaked the original recipe so that it wasn't quite as sweet, and then added a thin layer of this incredible frosting. These new brownies have since become a basic food group in our house.

Software applications are just like brownies—a basic application can be eminently satisfying all by itself. But you can do a lot more with Visual FoxPro than just write basic apps. In this section, I'm going to dive into the frosting (metaphorically speaking, unfortunately) and show you a number of advanced tools and techniques you can use to enhance your applications.



# Chapter 23

## Including ActiveX Controls In Your Application

**As hard as it's going to be to believe, there will be times when you just can't get Visual FoxPro to do something on its own. Instead of just throwing up your hands and disappointing your customer, it's time to extend Visual FoxPro's native capabilities through the use of ActiveX controls. In this chapter, I'll explain what they are, show you where to find them, and demonstrate how to use them in an application.**

Visual FoxPro joined the big scary world of components back in 1994 when ActiveX control (called "OLE control" back then) integration was added to the product for version 3.0. This is the ability for discrete applications to communicate with each other, and is provided through two mechanisms. One is ActiveX Automation, where one full-blown application can control or be controlled by another. This topic is the subject of an entire book, *Microsoft Office Automation with Visual FoxPro*, by Tamar E. Granor and Della Martin.

The other mechanism is to extend an application through third-party objects. These objects, or controls, can be placed in any application that supports the ActiveX specification. For instance, a graphics control written to the ActiveX specification could be placed in a Visual FoxPro form, a Visual Basic form, or any other design tool that is compatible with ActiveX.

ActiveX controls can add tremendous flair to your application. Indeed, you could build a Visual FoxPro application that consisted of little more than a wrapper around a robust and highly specialized control. For example, you could use VFP to act as a front end to a mapping ActiveX control, relying on VFP's data handling to grab information from the user and select values from a table. Then you'd send those values to the ActiveX control where all of the heavy duty lifting would be performed, and an end result would be returned by the control to VFP for presentation to the user.

### What is an ActiveX control?

An ActiveX control is a file (or files) that provides additional functionality to a Windows application. It usually takes the form of an .OCX or .DLL along with, optionally, other associated files. These files are usually located in the WINDOWS\SYSTEM directory on Windows 95/98, or the WINNT\SYSTEM32 directory on Windows NT.

You might also see a file with the same stem and a .DEP extension. This is a dependency file, which is simply a text file that contains information about the run-time requirements of an ActiveX control. For example, a dependency file might list which files are needed, how they are to be registered with Windows, and where they should be installed.

You might also find a file with the same stem and an .SRG extension. This is a license file that points to a key in the registry. You shouldn't have to mess with it at all.

Once you acquire an ActiveX control, you'll install it on your system much like you would any other Windows program. Then you'll typically have to perform some sort of "registration"

process so that your development tool, such as Visual FoxPro, knows that it's there. Once you've performed both of these steps, the ActiveX control will be available much like all other controls in your development environment.

ActiveX controls typically have a visual component because they are often intended to supplant or replace existing functionality. If you've worked with VFP grids, you can quickly come up with a list of enhancements you'd like to see—for example, intelligent headers, more variety in the control in each cell, intelligent resizing of column widths and row heights, the ability to highlight the current row and/or cell in a variety of ways, automatic parent and child displays in one grid, and so on. You can get an ActiveX grid control that has more functionality than the native VFP grid.

However, ActiveX controls don't always have to have a visual display. You can get an ActiveX control that performs communication tasks, either through a serial port or through the network. This control may or may not have a visual component—and even if it does, you might choose not to use it.

## **Where—and how—do I get them?**

ActiveX components can be found everywhere; indeed, that's become a problem. It seems that nearly every Windows product now comes with a handful of ActiveX controls.

### **In the box**

Microsoft Windows comes with several controls, and each of the visual tools in Visual Studio comes with its own set as well. If you have Visual Studio instead of just VFP, you should install Visual Basic if for no other reason than to install all of the controls that come with it.

I can't keep track of which controls come with each tool, but the Visual Studio online help lists which controls come with each product. **Tables 23.1** and **23.2** list the .OCX files that are installed with VFP and VB, respectively.

Okay, you're probably thinking, "Some of these sound cool, but where do I go for more information? How do I find them? How do I know if they're going to work with Visual FoxPro? How can I make sure my car doesn't get dinged in the shopping center parking lot when I go shopping next weekend?"

First of all, these are all found in the Windows SYSTEM or SYSTEM32 directory. You can also do a search on ".OCX" in Windows Explorer if you are looking for a specific control.

The VFP table nicely lists the help file for each ActiveX control, whereas you're pretty much on your own with VB. There isn't necessarily a 1-to-1 relationship between the name of an .OCX and its help file, because a single help file might cover multiple ActiveX controls.

You can find these tables in the MSDN online help under the topics "ActiveX Controls Overview" (for Visual FoxPro) and "ActiveX Controls File Names" (for Visual Basic).

As far as knowing whether a particular control is going to work with VFP, you're pretty much on your own. You should have a reasonable expectation that they will, but you can also bet on a few idiosyncrasies as well. The Microsoft controls are reasonably well-behaved—once you wander further afield, the possibility that you'll run into anomalous behavior increases.

**Table 23.1.** ActiveX controls installed with Visual FoxPro 6.0.

| <b>File</b>   | <b>Controls</b>                                                                                                                                                                                                                                                                                 | <b>Help File</b> |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------|
| COMCTL232.OCX | Animation control<br>Datetimepicker control<br>Monthview control<br>Updown control                                                                                                                                                                                                              | CMCTL298.CHM     |
| FOXHWND.OCX   | Visual FoxPro HWND Control                                                                                                                                                                                                                                                                      | FOXHELP.CHM      |
| FOXTLIB.OCX   | Visual FoxPro Foxtlib Control                                                                                                                                                                                                                                                                   | FOXHELP.CHM      |
| MCI32.OCX     | Multimedia MCI control                                                                                                                                                                                                                                                                          | MMEDIA.CHM       |
| MSCHRT20.OCX  | MSChart control                                                                                                                                                                                                                                                                                 | MSCHRT98.CHM     |
| MSCOMCTL.OCX  | ImageCombo control<br>ImageList control<br>ListView control<br>ProgressBar control<br>Slider control<br>StatusBar control<br>TabStrip control<br>Toolbar control<br>TreeView control                                                                                                            | CMCTL198.CHM     |
| MSCOMM32.OCX  | MSComm control                                                                                                                                                                                                                                                                                  | COMM98.CHM       |
| MSINET.OCX    | Microsoft Internet Transfer control                                                                                                                                                                                                                                                             | INET98.CHM       |
| MSMAPI32.OCX  | MAPI Message control<br>MAPI Session control                                                                                                                                                                                                                                                    | MAPI98.CHM       |
| MSMASK32.OCX  | Masked Edit control                                                                                                                                                                                                                                                                             | MASKED98.CHM     |
| MSWINSCK.OCX  | Winsock control                                                                                                                                                                                                                                                                                 | MSWNSK98.CHM     |
| MSWLESS.OCX   | CheckBox control (Lightweight)<br>ComboBox control (Lightweight)<br>CommandButton control (Lightweight)<br>Frame control (Lightweight)<br>HscrollBar, VScrollBar controls (Lightweight)<br>ListBox control (Lightweight)<br>OptionButton control (Lightweight)<br>TextBox control (Lightweight) | LWTCT98.CHM      |
| PICCLP32.OCX  | PicClip control                                                                                                                                                                                                                                                                                 | PICCLP98.CHM     |
| RICHTX32.OCX  | Rich Textbox control                                                                                                                                                                                                                                                                            | RTFBOX98.CHM     |
| SYSINFO.OCX   | SysInfo control                                                                                                                                                                                                                                                                                 | SYSINF98.CHM     |

**Table 23.2.** ActiveX controls installed with Visual Basic 6.0.

| <b>Component Name</b>                  | <b>File Name</b> |
|----------------------------------------|------------------|
| Microsoft ADO Data Control 6.0         | MSADODC.OCX      |
| Microsoft Chart Control 5.5            | MSCHART.OCX      |
| Microsoft Comm Control 6.0             | MSCOMM32.OCX     |
| Microsoft Common Dialog Control 6.0    | COMDLG32. OCX    |
| Microsoft Data Bound Grid Control 5.0  | DBGRID32.OCX     |
| Microsoft Data Bound List Controls 6.0 | DBLIST32.OCX     |
| Microsoft Data Repeater Control 6.0    | MSDATREP.OCX     |
| Microsoft Data Grid Control 6.0        | MSDATGRD.OCX     |
| Microsoft Data List Controls 6.0       | MSDATLST.OCX     |
| Microsoft FlexGrid Control 6.0         | MSFLXGRD.OCX     |
| Microsoft Grid Control                 | GRID32.OCX       |

| Component Name                               | File Name    |
|----------------------------------------------|--------------|
| Microsoft Hierarchical Flex Grid Control 6.0 | MSHFLXGD.OCX |
| Microsoft Internet Transfer Control 6.0      | MSINET.OCX   |
| Microsoft MAPI Controls6.0                   | MSMAPI32.OCX |
| Microsoft MaskedEdit Control 6.0             | MSMASK32.OCX |
| Microsoft Multimedia Control 6.0             | MCI32.OCX    |
| Microsoft PictureClip Control 6.0            | PICCLP32.OCX |
| Microsoft RemoteData Control 6.0             | MSRDC20.OCX  |
| Microsoft RichTextBox Control 6.0            | RICHTX32.OCX |
| Microsoft SysInfo Control 6.0                | SYSINFO.OCX  |
| Microsoft TabbedDialog Control 6.0           | TABCTL32.OCX |
| Microsoft Windows Common Controls 6.0        | MSCOMCTL.OCX |
| Microsoft Windows Common Controls-2 6.0      | MSCOMCT2.OCX |
| Microsoft Windows Common Controls-3 6.0      | COMCT332.OCX |
| Microsoft Winsock Control 6.0                | MSWINSCK.OCX |

## More? You want more?

Of course, like other gadgets, you can never have too many ActiveX controls. (Well, actually, you can, depending on the type of hardware and other system resources available on your machine—but that's a different matter.) Where should you go to find them?

The first place to start, as far as I'm concerned, is a place called [www.vbextras.com](http://www.vbextras.com). Run by Mike Schinkel, these folks have been in the biz for years and years, and they put out the nicest little (well, okay, 120 pages isn't exactly *little*) catalog dedicated to ActiveX controls and ancillary products. It's the rare individual who can leaf through the entire catalog without turning down the corner on at least three or four pages: "Oh, cool! I've gotta have that one! I don't know what I'm going to do with it, but I've gotta have it." The next place to look is [www.active-x.com](http://www.active-x.com). It's dedicated solely to ActiveX controls and will get you deep into the mix.

From these two sites, you'll be able to find links and search engines to just about anything you might want.

Once you find an ActiveX control of interest, you'll want to determine whether the ActiveX control will work with Visual FoxPro. There are four possible situations. The first is that the vendor has already tested and determined that it will. Controls from DBI Technology ([www.dbi-tech.com](http://www.dbi-tech.com)) and Microsoft fall into this category. The second is that the vendor will claim that it does, but they haven't actively tested it. The third category is that the vendor will indicate that they know positively that the control will not work in VFP. The fourth is the catch-all of "We don't know—why don't you try it out for us?"

Lest you think that I'm being negative about this situation, or just harping on it too much, let me reassure you that I don't mean to be—I'm just dealing with reality. The fact is that no ActiveX control is going to work with every development tool. The differences in the way each tool hosts an ActiveX control are large enough that the only ActiveX control that will work everywhere is a control that does nothing. And because the VB market is the largest, that's where most vendors concentrate their efforts. This is not to say that controls written specifically for the VB marketplace won't work elsewhere; it just pays to be cautious. Treat each new ActiveX control as Research and Development, not as a well-engineered, highly tested "gimme."

## Installing ActiveX controls in Windows

Once you've found a third-party ActiveX control of interest, you'll either have it shipped to you via snail mail, or (more likely) just download it from the Web. In either case, you'll eventually have some sort of installation package. You might have to execute or unzip the file you get from the vendor into a temporary directory to end up with an installation executable, along with, perhaps, a help file, a readme, and other stuff.

First back up the installation file (or files). As I mentioned in Chapter 16, I have a directory on my server called "Zip Archives" in which I store all of the tools I've downloaded or otherwise don't have physical media for anymore.

Next, run the installation routine. This should perform the following tasks: (1) copy the OCX (and any other related files) to the appropriate directory, such as the Windows SYSTEM or SYSTEM32 directory, (2) create a directory for the ActiveX control that includes documentation and samples, and (3) create a Windows Registry entry for the control.

This Registry entry is pretty important, because all Windows development tools will look for controls that are registered (with Windows) in order to know about them.

## Manually installing ActiveX controls in Windows

It is possible that the installation won't automatically register the control. If this happens, you can run into several situations that can be rather baffling.

The first is when you look for your brand new ActiveX control in your development environment, and you can't find it anywhere in the list of controls. If it's not in the list, it's not registered.

The second situation can occur when the control is initially registered, but then something on your machine changes, and the registration is removed. However, the control may still show up in the list of controls in your development environment for any number of goofball reasons. When you try to actually use the control in your development environment, the operation fails.

The third situation is when you try to run the application that contains the ActiveX control. Typically what happens here is that the installation routine fails to register the control automatically, and you click on through to finish the install, receiving one of those helpful "Installation failed" error messages. You might still attempt to run the application, the form will load partway, and then you'll get a message along the lines of "Error instantiating control. Record 26 of <name of VCX>." If you're not used to this, you'll end up opening the .VCX manually as a table, only to find out that Record 26 is the pointer to the ActiveX control on the form.

You might also encounter situations where you'll have to manually register it yourself. For example, suppose you're debugging or testing new versions of the control. It's pretty common for the vendor, instead of creating a whole new installation routine, to just send you a copy of the new .OCX and expect you to register the control yourself.

In any of these cases, you're going to want to (or, at least, need to) manually register the control with Windows. Here's how:

1. First, you'll need to know the file name of the ActiveX control. Hopefully you know this—either from the list of controls I provided earlier in this chapter, or from examining the installation package of the ActiveX control. You could even do a scan

of the Windows SYSTEM or SYSTEM32 directory before and after you did the automatic installation of the ActiveX control.

2. Put a copy of the control (along with any related .DEP or other files) in the SYSTEM or SYSTEM32 directory. Unfortunately, I can't help you much more in this step—you'll have to read the vendor's documentation (yeah, ha!) or talk to them to make sure you know all of the files needed.
3. Run the "Reg Server" program to tell Windows that a new ActiveX sheriff's in town. This program, located in the SYSTEM or SYSTEM32 directory, will actually create the appropriate entries in the Windows Registry so that the ActiveX control can be found by other applications and development tools. The syntax of the command, for a control named MYCOOL.OCX, is

```
REGSVR32 C:\WINNT\SYSTEM32\MYCOOL.OCX
```

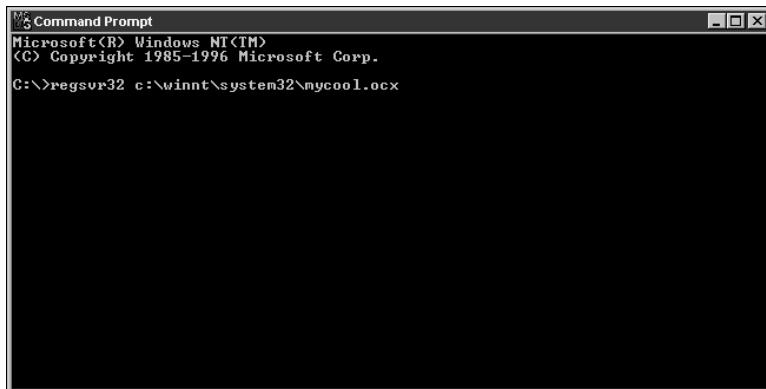
When the Reg Server program is finished, you'll get a dialog indicating success or failure. You can include a "/S" switch to run in "silent" mode—this way you won't get a dialog.

You can run this command from a Command Prompt (that's the 32-bit Windows version of what old-timers call a DOS box). Select Start, Programs, Command Prompt and you'll get a window with the Command Prompt interface. Ah, yes, the good old days. See **Figure 23.1**.

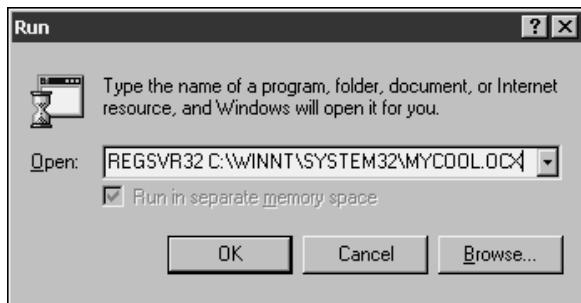
You can also use the Run menu option from the Start menu. See **Figure 23.2**.

There are all sorts of things that can go wrong here, and each of these problems is pretty annoying for a developer new to ActiveX, so let me discuss them.

First of all, Windows should know where REGSVR32 is located, so you shouldn't have to type a fully qualified path for the program. Nonetheless, if you've got an unusual configuration (that's French for "screwed up"), REGSVR32 might be installed where Windows can't find it, and you might have to include the full path.



**Figure 23.1.** Running REGSVR32 from the Command Prompt.



**Figure 23.2.** Running REGSVR32 with the RUN command.

Second, while Windows should know where the .OCX file is, assuming it's in the SYSTEM or SYSTEM32 directory, it might get confused if there are multiple copies of the .OCX file laying around. This can happen particularly if you are running REGSVR32 from a directory that contains a second copy of the .OCX. My advice: Use the fully qualified path name for the .OCX.

Third, you need to include the extension. I've been bitten many a time trying to register a component (particularly an .EXE component) without including the .EXE extension.

If you continue to have troubles with manually registering an ActiveX control, follow these steps (yes, every one of them):

1. Reboot your machine.

2. Uninstall (or unregister) the ActiveX control from your development tool.

3. Unregister the control using the following command:

```
REGSVR32 C:\WINNT\SYSTEM32\MYCOOL.OCX /U
```

4. Delete the ActiveX control from the SYSTEM or SYSTEM32 directory.

5. Search the Windows Registry for any traces of the ActiveX control. Using the Start, Run menu option, run RegEdit. Then select the Edit, Find menu command inside the Registry Editor dialog. Remove those traces if found.

6. Reboot your machine.

7. Reinstall the ActiveX control by copying it to the SYSTEM or SYSTEM32 directory and then registering it with Windows again.

If these steps don't work, you might have a corrupt version of the control.

## How do I put them into my development environment?

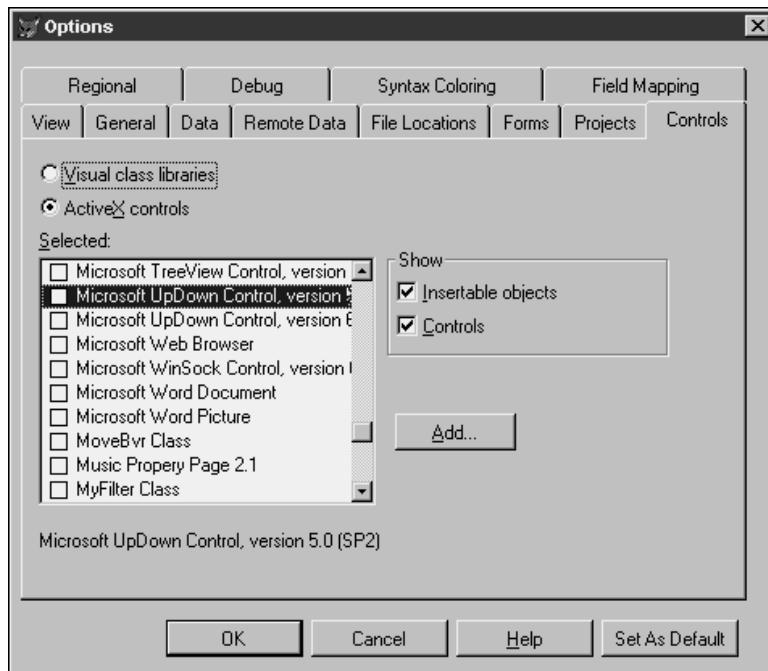
In the previous sections, I've referred to "registering the control with your development environment." You actually don't always have to do this. It depends on which mechanism you want to use when adding ActiveX controls to a form. Yes, just like everything else, there's more than one way to skin a Fox.

The first mechanism involves dropping ActiveX controls onto a form from the ActiveX Controls toolbar, while the second involves dropping a standard VFP ActiveX Control base class (or your own subclass, of course) onto a form from the Form Controls toolbar and then selecting which ActiveX control is to be associated with that class. You don't need to do anything special if you choose the second mechanism, but you do for the first.

To select ActiveX controls that will be displayed on the ActiveX controls toolbar, open the Tools, Options dialog and select the Controls tab, as shown in **Figure 23.3**. You can display a list of visual class libraries or a list of ActiveX controls by making the appropriate choice in the option button group in the upper left corner of the tab. One slight annoyance is that the description in the list box control doesn't display completely in the list box—in Visual Basic, you can also scroll the list box to the left and right. You can see the full name of the highlighted ActiveX control below the list box, but that's small consolation. Given the responsiveness of the Fox team, I'll bet we see this fixed in a future version of Visual FoxPro.

This is important because some controls have two versions, and you can't always tell which one you're selecting. In Figure 23.3, the highlighted control is the Microsoft UpDown Control, version 5.0, while in **Figure 23.4**, version 6.0 is highlighted.

On my system, I've got 191 ActiveX controls registered with Windows. Obviously, you wouldn't want all 191 on your ActiveX controls toolbar—so just select the controls you want to see by checking the check box to the left of the name in the Selected list box, as I did in Figure 23.4.



**Figure 23.3.** The Controls tab of the Tools, Options dialog, showing version 5.0 of the UpDown Control.

If the control you want to select isn't in the list, it might be on your machine but not registered with Windows yet. You don't have to leave Visual FoxPro to register the control—if you click the Add button on this tab, you'll be able to point to an .OCX file. Doing so will automatically register the control with Windows and add it to the list box at the same time.

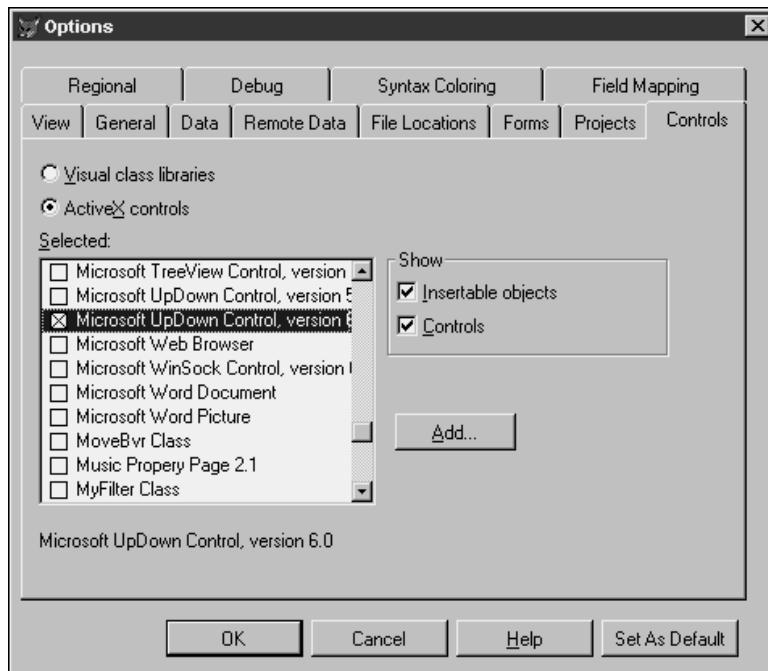
Once you've selected the controls you want, close the dialog, selecting Set As Default if desired. Next, open the Form Controls toolbar, click the View Classes button in the toolbar (the one with the books and the small arrow in the lower right corner), and select the ActiveX controls menu option as shown in **Figure 23.5**.

Once you select the ActiveX controls menu option, the Form Controls toolbar will change to display just the selected ActiveX controls as shown in **Figure 23.6**.

## How do I use them in an application?

As I mentioned earlier, there are two ways to drop an ActiveX control on a form. Having seen how to populate the ActiveX Controls Form Controls toolbar in the previous section, it's probably pretty obvious how to proceed. You create a form and pop the toolbar's ActiveX control on the form.

While this method is easy to use, it involves more setup. The second mechanism is the opposite—there's no setup required, but each time you drop a control on a form, you have to go through more steps.



**Figure 23.4.** The Controls tab of the Tools, Options dialog, showing version 6.0 of the UpDown Control.

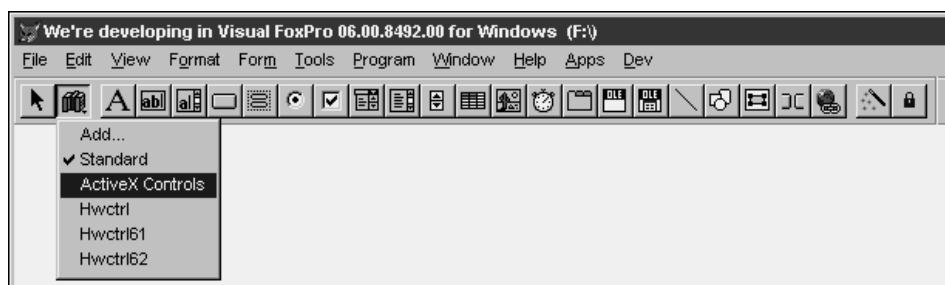
First, you'll drop an ActiveX Control on the form in question. The ActiveX Control is found in the Standard Form Controls toolbar, as shown in **Figure 23.7**.

Once you drop it on a form, you'll get an Insert Object dialog as shown in **Figure 23.8**.

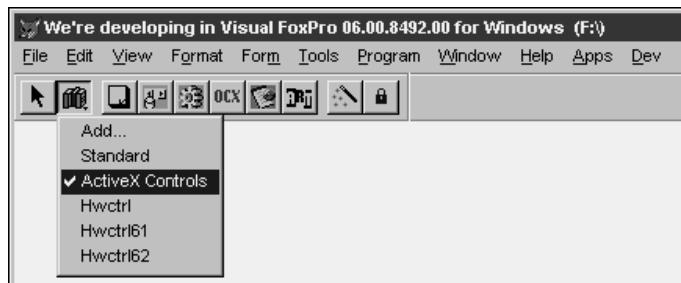
You'll select the ActiveX control you want in the Control Type dialog, and then click OK. The end result will be the same as if you dropped the ActiveX control onto the form from the ActiveX Controls Form Controls toolbar. See **Figure 23.9**.

You can actually go a bit further, and create subclasses of the ActiveX controls you want to use, and then place those subclasses in their own class library, which means they'll all be together on their own toolbar. If you make extensive use of ActiveX controls, this might make a lot of sense so that you can keep groups of commonly used ActiveX controls together on the same toolbar.

There are other benefits to subclassing your ActiveX controls. For example, you can add your own standard properties and methods to your class definition, and then every time you create an instance of the control, it will automatically include your additions. You can also create an easier to use interface than that which is exposed by the ActiveX control itself. My tech editor has subclassed the CommonDialogs control and added properties that are easier to set than adding a bunch of numbers together and putting them in the Flags property. You could even add extra functionality.



**Figure 23.5.** Selecting the ActiveX Controls toolbar from the View Classes button in the Form Controls toolbar.



**Figure 23.6.** The Form Controls toolbar with a half-dozen ActiveX Controls buttons.

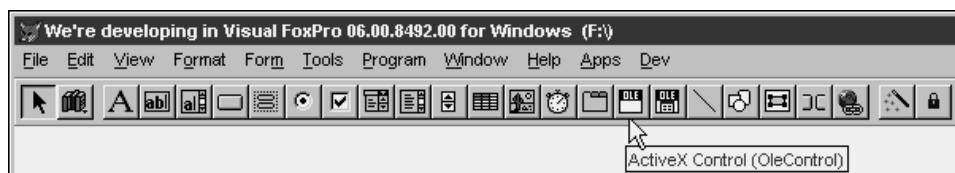
Once you've got the ActiveX control on the form, there's one more step involved—actually interfacing the control and the rest of your form. How that works depends completely on the control you're using. You'll need to refer to the help file for specifics. Hopefully you'll have sample code—even possibly in Visual FoxPro, but more likely in Visual Basic—so you can follow along in someone else's footsteps.

## How do I ship them with an application?

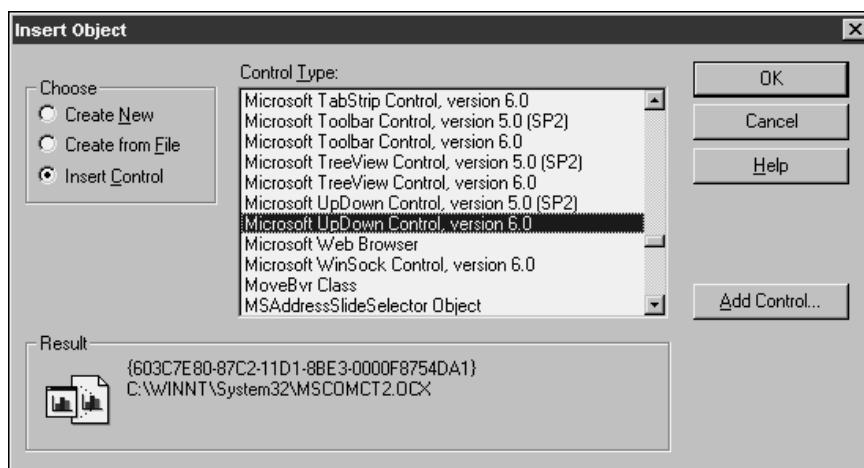
So it's been awhile since the previous section. You've gotten the controls to interface with the rest of the form, you've had a few glitches, or maybe a lot, but you're finally done. The miracle has occurred, and you're ready to ship. What do you now?

It seems pretty logical that because Windows is all integrated, you'd be able to create a build directory containing your executable and a few data files, and then run the Setup Wizard to create a set of installation files. The Setup Wizard ought to be smart enough to find any ActiveX controls in your project and include them with your application's installation files.

Prior to version 6.0, this wasn't the case. You had to copy all of the necessary files associated with your ActiveX control to the directory from which you were going to build your installation files, and then specify which files were ActiveX controls (and some other stuff) during the setup routine.

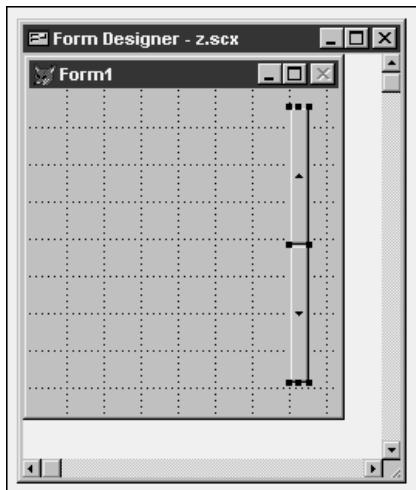


**Figure 23.7.** The ActiveX Control on the Form Controls toolbar.



**Figure 23.8.** The Insert Object dialog.

With VFP 6.0, however, the Setup Wizard will take care of all of this for you. All you have to do is make sure that the appropriate controls are installed properly on your machine. The Setup Wizard will then do everything else. See a complete example in Chapter 26, “Distributing Your Applications with the VFP Setup Wizard.”



**Figure 23.9.** Form Z after dropping the UpDown Control on it.

## What if it doesn't work?

Visual Basic programmers have had it very, very good in one respect of development—their tool was designed and built with the assumption that developers would not be relying on native VB, but instead would be extending VB through the use of third-party controls. As a result, many of the most innovative interfaces you see in the Windows world are created with VB.

Visual FoxPro, as does Visual Basic, comes with a set of standard controls—command buttons, option groups, check boxes, list boxes, spinners, and so on. However, Visual FoxPro developers are used to relying on this group of controls, and thus haven't generated as much market demand for additional controls.

The upshot of this situation is that most ActiveX controls are designed and built with Visual Basic developers in mind. Thus the developers of those ActiveX controls don't always consider the Visual FoxPro market, and if they do, they often just give us a passing nod. It has historically been somewhat of a crapshoot whether a specific control will work—or work well—with Visual FoxPro.

If you run into a specific control that seems to be misbehaving, contact the manufacturer. Explain what you're trying to do, and offer to work with them. It takes a lot more time, to be sure, but (at least in my limited experience) vendors are often willing to do something to resolve the problem if they have someone to work with. Many ActiveX control vendors simply

---

don't know anything about VFP, and thus they're limited in terms of what they can test in the first place.

## DLL hell

ActiveX controls, for all the benefits they bring, suffer from one serious architectural flaw that Microsoft has been wrestling with for quite a while. As I've said, all ActiveX controls should be in the Windows SYSTEM or SYSTEM32 directory. (Once in awhile, some misbehaving application places its .OCXs elsewhere.) But the idea is by keeping all of the components in a single location, many applications will be able to leverage the same files.

Yes, that's right—instead of having 19 spellcheckers and four zip-code-lookup programs and a dozen pop-up calendar controls scattered all over a user's computer, the ActiveX architecture would allow all 19 programs requiring a spellchecker to use the single copy in SYSTEM or SYSTEM32. And those dozen applications that used a calendar control would all reference the same .OCX as well. This cuts down on the clutter and the wasteful duplication that was starting to occur.

Even better, of course, was the idea of reusable components—you didn't have to write your own zip-code-lookup object; you could simply reuse one that was already built, tested, and installed. Pretty soon, the theory went, all applications would be simply assemblies of mounds and mounds of components. What a sweet deal!

Of course, reality bites.

What has happened is a nightmare of incompatible versions stomping on each other and not working with specific versions of other controls. Here are a couple of examples.

You've built an app using a TCP/IP ActiveX control. You install it on a user's machine, and it's running fine. One day, your user calls you up complaining that it doesn't work anymore. "What changed?" "Nothing," they say. Well, one thing did change—someone else installed a new version of another application that installed an old version of that TCP/IP control used by your app. Big surprise—your app can't use the old version because it requires access to functions that weren't in the old version. So your app is broken, and the user thinks it's your fault.

But it can get worse. Suppose you've built an app using a graphics ActiveX control. Naturally, your app is running fine until the user decides to update their copy of another application. This time, however, the upgrade installs a newer version of the same graphics ActiveX control. And, of course, it's not compatible with your application because it's been changed, just ever so slightly—a method was renamed, or a property's value changed from numeric to character. And again, your application is broken, and the user blames you.

What's the resolution to this situation? There really isn't any. Microsoft is well aware of the situation, and claims to be devoting significant resources to a solution. However, that doesn't help you or me now as we're trying to ship applications.

Many people have developed workarounds. One is a tool used throughout the VB world called Version Stamper ([www.desaware.com](http://www.desaware.com)) that can help to a degree. But the bottom line is that this is a serious problem that, fortunately or unfortunately, most Visual FoxPro developers aren't exposed to often. It is a significant problem for the rest of the Windows world, however, and you need to be aware of it when you start using ActiveX controls.



# Chapter 24

## Extending Visual FoxPro Through the Windows API

Sooner or later you're going to need to do something that isn't native to Visual FoxPro, but that *is* readily available in Windows. Microsoft has provided a rich set of functions inside Windows, and they've given us an API—an Application Programming Interface—so that we can get at those functions from inside VFP itself. It's not quite as simple as just calling a new set of functions, so in this chapter I'll explain how to access those functions and what to do with them.

Many hard-core Visual FoxPro programmers have shied away from the Windows API because, well, it felt too much like programming in C. "If I wanted to mess with structures and variants and declarations and pointers, I'd be geeking along in C++ instead of VFP." It's really not too bad, though—after you learn a couple of basic concepts and tips.

### What is the Windows API?

When you write a program, you are essentially talking to the machine, but with a number of layers in between. The first layer is the operating system; unless you're writing in machine code, you use the operating system to talk to the machine. You do so by using the operating system's API. Instead of letting a programmer loose on the whole machine, the API is structured as a set of function calls that allow you to do things in a structured, and, hopefully, failsafe manner. But writing a program through API calls is tough. Instead, wouldn't it be nice if someone packaged a bunch of these API calls together? For example, instead of having to issue a series of five or six functions in order to open a file on disk, what if you could do so with just one function, and what if you called it "USE." Yeah, you get the idea. So you can think of VFP (and other high-level programs like Delphi) as organized collections of function calls to the operating system API.

Because Windows is pretty near an operating system all by itself, it too has an API, and it's pretty darn robust. More robust, in fact, than VFP itself. So that means while you can do a lot from within VFP, there's still some stuff for which you're going to have to use the API.

Windows is, in a sense, a collection of .DLLs, and just as you can call functions from .DLLs that you have written yourself, you can call functions from the Windows .DLLs. There are five primary .DLLs that are known as the Win32API:

- USER32.DLL
- KERNEL32.DLL
- GDI32.DLL

- MPR.DLL
- ADVAPI32.DLL

When you call a function from one of these, you just have to specify that it's coming from the Win32API. If you call a function from other .DLLs, you have to name that specific .DLL.

## What do functions look like in the Windows API?

Here are six examples of API functions:

- CascadeWindows
- ClosePrinter
- GetActiveWindow
- GetFileVersionInfo
- GlobalMemoryStatus
- LoadImage

In each case, you can pretty much guess what the function is going to do, can't you? But you're probably wondering where I dug these up, and how I knew about them. Some of these (together with the genesis of some of the examples) came from a series of articles that Gary DeWitt wrote in *FoxTalk* in late 1997 and early 1998, while others I found on my own by spelunking through the WIN32API.TXT file. The *what?* you say?

Every function in the WIN32API, for example, is described in a text file called, funny enough, WIN32API.TXT. This file is buried in the Visual Studio hierarchy:

`Program Files\Microsoft Visual Studio\Common\Tools\Winapi`

The Win32API is not the only API that has a text file that describes its functions. But digging through any of these text files is pretty bad news. Because you're a Visual Studio kind of developer, you can use the Visual Basic API Viewer application to put a friendly user interface on the file.

## Using the API Viewer

The API Viewer application enables you to browse through the declares, constants, and types included in any text file. You can get to it "manually" by just running APILOAD.EXE (it's in the same directory as the .TXT file—Microsoft Visual Studio\Common\Tools\Winapi), or you can get to it through Visual Basic. Here's how to run it in VB:

1. Load VB.
2. Select the Add-Ins, Add-In Manager menu option to open the Add-In Manager dialog as shown in **Figure 24.1**.
3. Select the VB6 API Viewer in the Available Add-Ins column of the list box.

4. Check the Loaded/Unloaded check box. Check the Load on Startup check box if desired.
5. Close the Add-In Manager dialog.
6. Select the Add-Ins, API Viewer menu option to open the API Viewer dialog as shown in **Figure 24.2**.
7. Select the File, Load Text File menu option, and select the WIN32API text file.



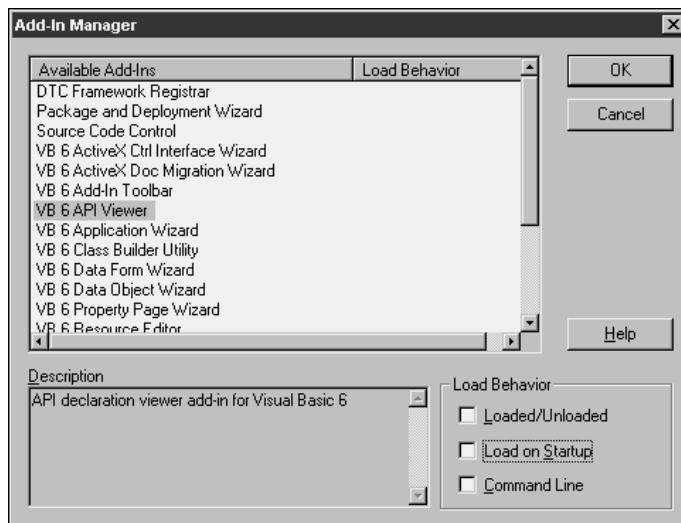
*By selecting View, Load Last File, you can have the API Viewer automatically display the last file you viewed in it.*

8. Select the type of item you want to review from the API Types combo box.
9. Type the first few letters of the API function you are interested in, or just scroll through the Available Items list box until you've found the item you want.

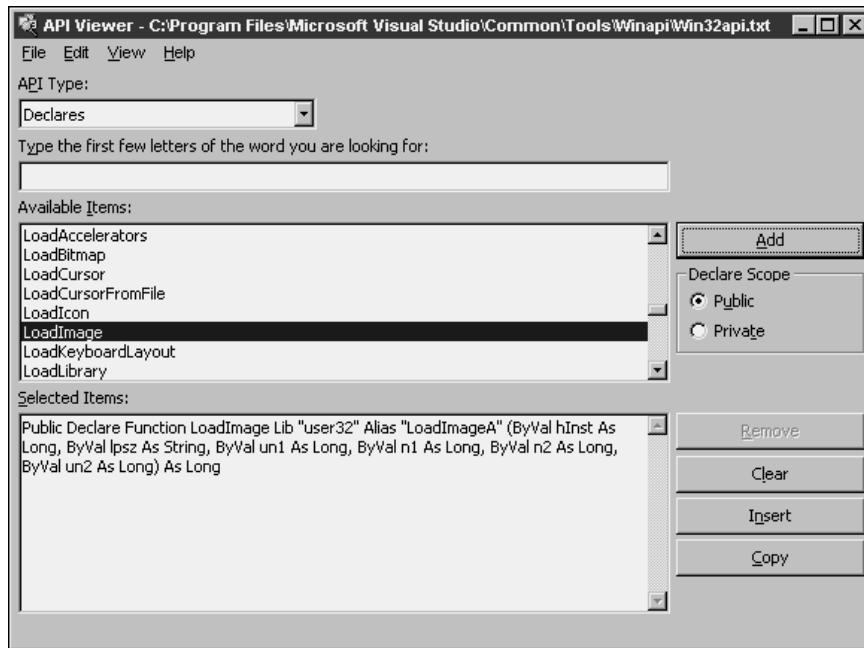


*Just scrolling through the list is a good way to get a general feel for all of the types of function calls available—and that's just in the Windows32 API!*

Although this list is intellectually intriguing, it's not all that useful, because there is no description of any API functions. How do you find out? The best way is to pick up a copy of Dan Appleman's *Programmer's Guide to the Win32API*. Yes, true, it's a VB book, but it's also been the bible for the Win32API for the better part of a decade. You'll be able to scan through and find out what each of these functions does. Unfortunately, all of the examples are in VB (*there's a surprise!*) so you've got a bit more work to do.



**Figure 24.1.** The Add-In Manager in Visual Basic 6.0.



**Figure 24.2.** The API Viewer in Visual Basic 6.0.

## How do I access a WinAPI function?

Just because you have the file describing all of the functions—and a copy of Windows—doesn't mean you can do anything in VFP yet. First, you have to get VFP to recognize the existence of the function. This is just like having a back-end database on your disk: Just because it's there doesn't mean VFP knows it's there—you have to create a connection to it!

In order to get VFP to talk to the Windows API, you use the `DECLARE ... DLL` function. Here is the basic syntax for this function:

```
DECLARE <return type> <function name> in <DLL> <parameters>
```

For example, the VFP command:

```
DECLARE INTEGER GetActiveWindow in WIN32API
```

tells VFP that it should now recognize a function called `GetActiveWindow`, that this function returns an integer value, and that the function is in the `WIN32API`. Similarly, the following VFP command tells VFP that it should now recognize a function called `GetFileVersionInfo`, that this function also returns an integer, and that the function is in the `VERSION.DLL`. Furthermore, this function takes four parameters, two strings sandwiching a pair of integers:

```
DECLARE INTEGER GetFileVersionInfo in VERSION ;
      STRING @lpstrFilename ;
      INTEGER ;
      INTEGER dwLen ;
      STRING @lpData
```

These might seem daunting, but they're really not too bad. I'll explain what the various return value and parameter expressions mean shortly. But first, to kind of bring this explanation full circle, I need to explain how you use an API function once it's been declared.

## Once I've called it, how do I use it?

Once you use the DECLARE statement, all you've done is tell VFP that there's a new function in town. It's available for you to use just like any other VFP function, as long as, of course, you obey the rules. Because you're dealing with the API itself, you can potentially be messing with memory, addresses, and other things that don't behave elegantly when mishandled. Instead, mistakes here just GPF your machine and cause your local NFL team to lose next Sunday.

How about that!

There are basically two ways that you can use a Win32API function. The first is when a function returns a value. In that case, you simply call the function as if it were a VFP function, and assign the return value to a VFP memory variable. For example:

```
lnHandle = GetActiveWindow()
```

assigns the value of a window "handle" that was generated by the GetActiveWindow API function to a Visual FoxPro memory variable named lnHandle.

The second way, and the way you'll access most functions, is to define VFP memory variables and then call the API function, passing those memory variables by reference. When the call to the function is done, the memory variables that you had predefined will now be populated with values from the API function doing its thing. You're now free to use those values as you see fit. For example, to access properties of a window to which you've previously gotten a handle, you would use code similar to this:

```
lcCaption = space(255)
lnSize = len(lcCaption)
lnSize = GetWindowText(lnHandle, @lcCaption, lnSize)
if lnSize > 0
  wait window "The caption is " + lcCaption
else
  wait window "There is no caption for this window."
endif
```

There's a twist in this function call—you'll notice that the second parameter, lcCaption, has an "at sign" (@) in front of it. That's because I'm not passing a string of 255 spaces to the GetWindowText function; instead I'm passing a pointer to the address where those 255 spaces are stored in memory.

Many API calls don't pass values to the function; instead, they pass pointers (memory addresses to those values) to the function. If you're thinking this is sounding a lot like C, you're right, because that's how C works as well.

So the internals of the GetWindowText function see three arguments. Two are values—the handle of the window and the size of the string. The other is a pointer to the spot in memory where the Caption will be placed. The GetWindowText function is simply going to write to that spot in memory, up to a limit of 255 characters.

If you're tempted to go out at this point and start slinging code, you'd better wait. There are two very serious hidden gotchas just waiting to bite you.

## Structures

The previous example used simple variable types that are found in both Visual FoxPro and C—there were two integers, and the pointer to lcCaption was pointing to a text string. But C has another type of data—a structure—that has no mapping in VFP. You need to handle structures a bit differently.

You can envision a structure as an array designed by Picasso. It's a way to represent a group of data items that are each possibly different but still all related. For example, you might have cruised through the API Viewer and found the GetWindowRect function. That looks pretty darn similar to the GetWindowText function that I showed you a few paragraphs ago. So then you'd compare the VB definitions of GetWindowText and GetWindowRect:

```
Public Declare Function GetWindowRect Lib "user32" Alias "GetWindowRect"
    (ByVal hwnd As Long, lpRect As RECT)
    As Long
Public Declare Function GetWindowText Lib "user32" Alias "GetWindowTextA"
    (ByVal hwnd As Long, ByVal lpString As String, ByVal cch As Long)
    As Long
```

The GetWindowText function has a definition of the second argument “as RECT” but other than that, it looks about the same. So you might have concluded that you could call GetWindowRect just like GetWindowText, using a memory variable luRect (“u” because you don't know exactly what it is yet):

```
luRect = space(255)
lnSize = GetWindowRect(lnHandle, @luRect)
```

You might have figured you'd take a look at the luRect memory variable to see just what was in it after it was populated by the GetWindowRect function.

The problem is that the lpRect argument is a pointer not to a string, but to a structure, and that structure consists of four LONGs (four-byte integers). You can fool the Windows API into accepting luRect if you define it ahead of time properly—in this case, as a 16-byte string of zeros:

```
luRect = replicate(chr(0), 16)
```

This is done because a structure is a single continuous block of memory, and the only way we can map a block of VFP memory to a structure is to use a string—which is also a single continuous block of memory.

Putting all of this together, the following code grabs a window handle, determines the window caption, and determines the left, top, right, and bottom values that define the window:

```
DECLARE INTEGER GetActiveWindow in WIN32API
DECLARE INTEGER GetWindowText in WIN32API ;
    INTEGER, ;
    STRING @, ;
    INTEGER
DECLARE INTEGER GetWindowRect in WIN32API ;
    INTEGER, ;
    STRING @
lnHandle = GetActiveWindow()
lcCaption = space(255)
lnSize = len(lcCaption)
lnSize = GetWindowText(lnHandle, @lcCaption, lnSize)
luRect = replicate(chr(0), 16)
lnDummy = GetWindowRect(lnHandle, @luRect)
```

The one final fly in the ointment is that the string memory variable that was populated with values from the API structure isn't all that straightforward to deal with. The 16-character string comes back as a 16-byte binary value, which then needs to be broken into 4-byte sections, and then those sections need to be converted from strings to numeric values. It's not a big deal, so you can write a function that would do so, and call that function each time you need it.

For example, suppose you called the function Convert. Once luRect was filled, you could extract the left position of the window like so:

```
lnLeft = convert(substr(luRect, 1, 4))
```

Other structures aren't as cleanly defined. For example, the GetFileVersionInfo function fills a structure of varying size. That's right—you can think of this structure as a variable length field. How do you find out when one item in the structure ends and another begins? You look for it! Specifically, certain text strings, such as "Company Name," are included in the structure. Also, each item is separated by one or more CHR(0)'s. Thus, you simply grind out standard text-parsing code that looks for the known text strings and the CHR(0) demarcates, and grab the data in between.

There's a great little class by Christof Lange that handles converting to and from structures. You can find it on the Universal Thread at [www.universalthread.com](http://www.universalthread.com).

## Pointers as return values

The other gotcha I mentioned a bit ago has to do with the fact that the value that was returned by the GetActiveWindow API function was an integer—simply a handle to the window. What if the function returned a value that didn't map to a VFP data type? For example, some API calls return pointers. If you're calling that API function from C, you're in great shape, because C knows how to handle pointers as memory variables. But VFP doesn't. So just don't do that! There aren't many of them, but you do have to watch out for them.

## Conclusion

This chapter has just touched the surface of the Windows API. Whole books have been written about it, and whole programs have been written using it! But I've found that the first couple steps into the Win32API—because it feels so much like C—are the scariest. Once you've tried a couple on your own, the rest come considerably easier.

# Chapter 25

## Adding HTML Help to Your Applications

Online help has become pretty much a given these days. The previous version of Windows Help, WinHelp, was tough to implement and not that useful. The new version, HTML Help, is wonderful. In this chapter, I'll show you how to use HTML Help Workshop and how to incorporate an HTML Help file with your application.

I make no bones about it. I *like* HTML Help. A lot of people have made fun of me for it, but I think it's one of the nicest features in Visual FoxPro 6.0—the ability to include a .CHM file with your application. And even better than me liking it, users like it a lot, too!

I was about 80% done with a sizable (14 MB .EXE) VFP 6.0 application, named TWISTOR, when we started addressing the issue of help. This app had been a rewrite (and major upgrade) of an old FoxPro DOS system. My customer had just assumed that they'd do the same for help as they did last time—hold a few training classes and include a printed manual. Unfortunately, this system was so much larger and more complex that a training class wasn't going to be sufficient, and the complete manual would have ended up being 500 pages.

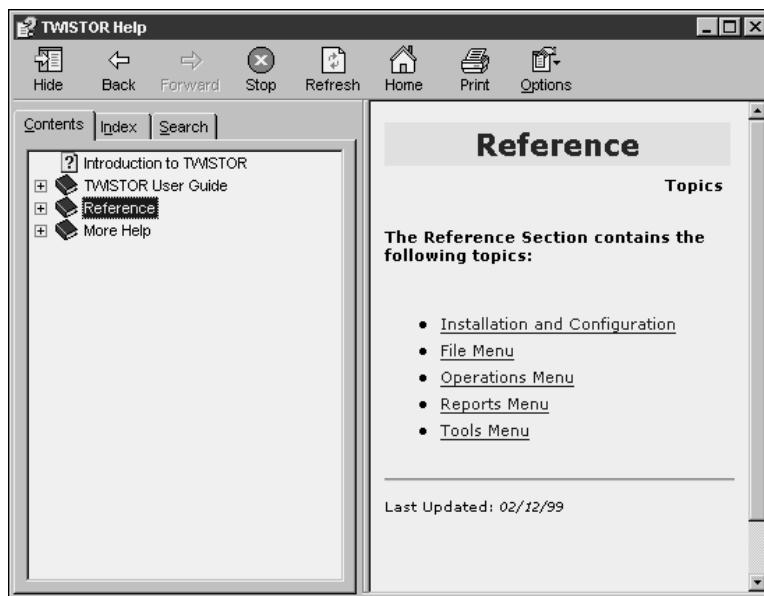
I suggested online help, but they scoffed because their experience with online help was less than positive. Then I showed them what an HTML Help system could look like. They oohed and aahed. Then I showed them the search and index capabilities. They were sold. The most telling comment of the meeting was, "I would even use this help system myself!" When I was done building their online help file, we had 293 help topics covering more than 500 pages, yet it was always available to anyone using the system, it was much more flexible than a manual, and it could be updated quickly and easily.

In this chapter, I'm going to show you how to be a hero with your users by providing a kick-ass help system based on HTML Help.

### What is HTML Help?

I'm tempted to respond to this question with a tart, "HTML Help is wonderful, that's what it is!" answer, but I've decided to let you come to this conclusion yourself.

The easiest way to explain HTML Help is to show you an example of it. If you call up the help file for the application I just mentioned, you'd see a screen as shown in **Figure 25.1**. While VFP (and Visual Studio) help is also an HTML Help file, it is rendered by a Microsoft custom engine that we don't have access to. The special engine includes the menu bar and some other capabilities. Thus, to show you what you can ship to your customers, I'll demonstrate through an application of mine.



**Figure 25.1.** The TWISTOR help file—in HTML Help format—contains three main sections.

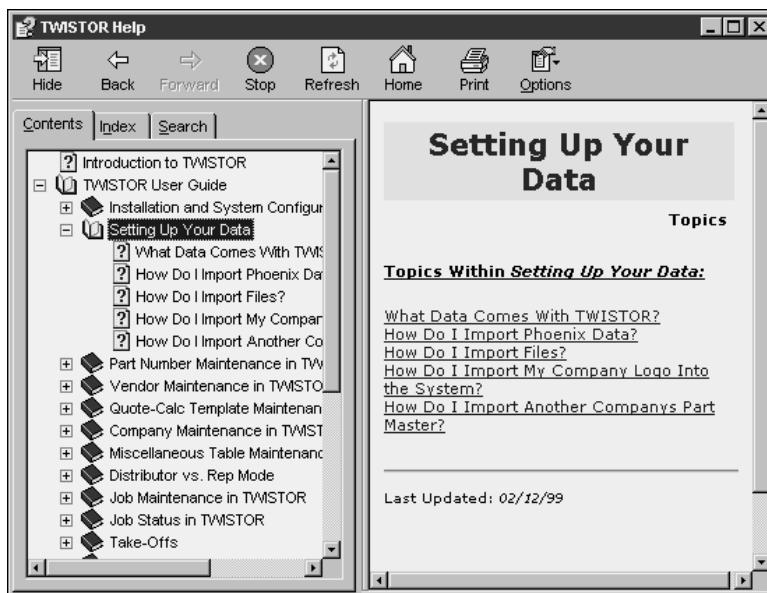
As you can see, I've divided this help file into an introduction and three sections. The Introduction is the “executive summary” that you'd typically put in a proposal or specification; for me, it's just a one- or two-page document. It goes to the level that you'd explain at a cocktail party if someone asked you (with real interest), “So what does TWISTOR do, anyway?”

The User Guide is the “How To” section of the manual and is subdivided into a number of functional areas. Each area then has one or more topics. Each of these topics contains step-by-step instructions and an explanation of the context of the instructions for a specific function.

In **Figure 25.2**, I've expanded the User Guide section to show each of the functional areas, such as Installation and System Configuration, Setting Up Your Data, Part Number Maintenance, and so on.

Under the functional area of Setting Up Your Data, there are specific topics, including an explanation of the data that initially ships with TWISTOR, and then a series of instructions for moving data from other systems into TWISTOR.

The left pane contains a TreeView control that operates like other Explorer interfaces you've worked with. Each section that has a (+) or (-) sign can be expanded or contracted to drill down through a hierarchy of sections, subsections, and topics. Although I've displayed just two levels, you can go deeper—indeed, you are more likely to start confusing users with a plentitude of levels than are to run into some limit.



**Figure 25.2.** The TWISTOR help file, expanded to show its functional areas.

If this were all that HTML Help did, it'd be pretty swell. I really like the hierarchical organization that allows you to drill down as need be. It's easy to go through a manual and get lost trying to remember which heading or subheading you're following unless the graphics people were blatant about heading styles: "THIS IS A LEVEL ONE HEADING" and "this is a level two heading." Because you can expand multiple levels at will, you can get a bird's-eye view a lot quicker.

Furthermore, having a topic page in the right pane tends to keep the discussion condensed—if it gets too long, it's hard to read in a single-topic page, and thus cries out for being broken up.

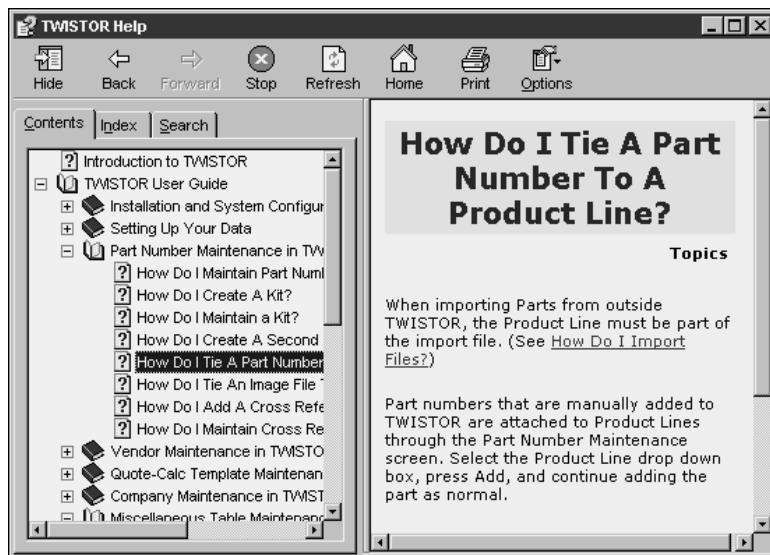
But there's more than just a hierarchical view of your help file. The grayscale images in this book don't display hyperlinks well, so it's a bit hard to see, but each of the topics in the right pane of the Setting Up Your Data topic in Figure 25.2 is actually a hyperlink to each topic. And, of course, you can embed hyperlinks in text as well, as shown in **Figure 25.3**.

## Index and Search tabs

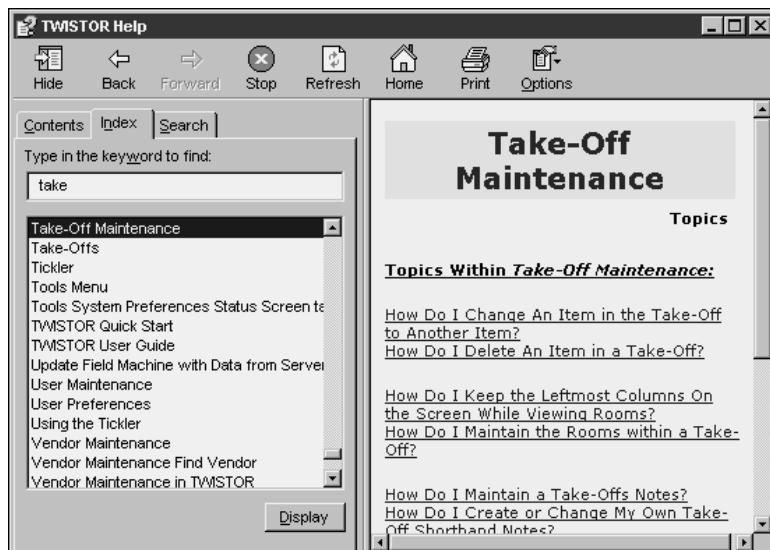
It's often not enough just to drop a big help file on your users' desks. You want to give them the ability to find stuff easily. The Index and Search tabs allow you to do so without a lot of effort on your part.

You can create a list of index entries that point to specific topics. The user then enters part of a keyword in the text box above the index entries list box. Entries matching the keyword will be displayed in the list box and the user can select the appropriate topic, as shown in **Figure 25.4**.

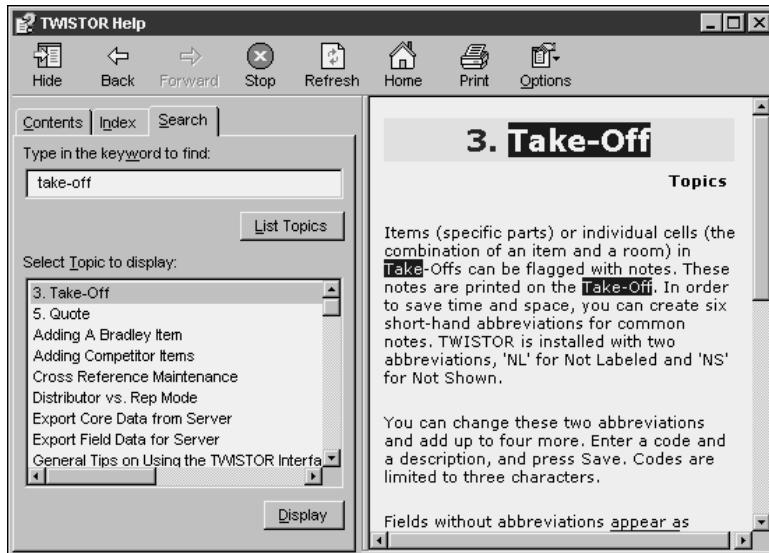
You can also let the user perform a full-text search of the help file by using the Search tab. Similar to the Index tab, the user will enter a keyword in the text box, click the List Topics button, and a list of topics containing the keyword will appear in the list box below. Selecting an entry in the list box will display the topic in the right-hand pane, as shown in **Figure 25.5**.



**Figure 25.3.** You can embed hyperlinks to other topics inside topic text.



**Figure 25.4.** Searching the index for take-offs.



**Figure 25.5.** Searching for the phrase “take-off” in the TWISTOR online help.

So that’s what the user sees. What’s underneath? What does the plumbing look like?

An HTML Help file consists of a single binary file on disk with a .CHM (pronounced “chum” by ‘softies and “chim” by the rest of us) extension. In order to run it, the HTML Help rendering engine (HH.EXE) and some ancillary components must be installed on the machine your application is running on. In a minute I’ll discuss what’s entailed in including these files in your application in order to ship it.

To create a .CHM file, you use a tool called HTML Help Workshop. You add a series of HTML files (yes, like those on the Web) to it, much like you add files to the Project Manager in Visual FoxPro, organize them into levels and sublevels, set a bunch of options, create an index, mark search keywords, and compile the results into a .CHM file.

## Creating an HTML Help file using HTML Help Workshop

HTML Help Workshop (HHW) is a Microsoft tool that comes with Visual Studio and is available for download from the Web. HHW is updated fairly regularly, so if you don’t want to wait for the latest Visual Studio Service Pack, or if you just want to work with HHW by itself, you should pop up to the Web and grab the latest version.

## Where is HTML Help Workshop?

I was going to try to show you where to go, but, of course, the download site has changed twice since I last downloaded an update, and I can’t find it as I’m doing this chapter. Your best bet is to hit [msdn.microsoft.com](http://msdn.microsoft.com) and click on the Downloads link. You’ll probably have to log on, or sign up, or plug in, or something.

Eventually, I hope, you'll find a file called HTMLHELP121.EXE or something similar, and install it. The version I'm running right now is 1.2 (4.73.8302.0), whatever that means—I've heard that 1.3 might be out soon.

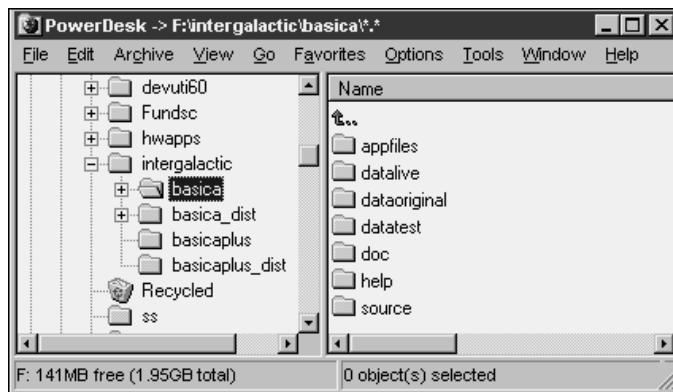
## Before you create your Help project

Before you run the HTML Help Workshop program itself, you'll want to prepare a couple of things. The first is a directory where your help project will live. I'd recommend a Help directory under the application directory—on the same level as your Documentation directory as described in Chapter 16. See **Figure 25.6**.

After you've set up your directory, you'll also want to gather all the files to be used in your HTML Help file. These files must be HTML files, and there are about a thousand different ways you can create them. I've always written the help file in Microsoft Word, and then created HTML files from within Word. Depending on which version of Word you're using, you'll have varying degrees of success with this.

I've found that Word 97 is much friendlier than Word 2000 because it just goes ahead and creates an HTML file and extracts any images it needs into files named IMAGE1, IMAGE2, and so on. Word 2000 tries to be clever, creating an HTML file and a pair of subdirectories, one containing image files that were extracted from the Word document, and another that contains a bunch of XML pointers. Unfortunately, I think there are still some bugs in this Word 2000 process—we continue to have problems converting chapters in our books to HTML files that can be successfully read by HTML Help Workshop. Of course, it might be a bug in HTML Help Workshop that can't handle the new Word 2000 files—hard to say at this time.

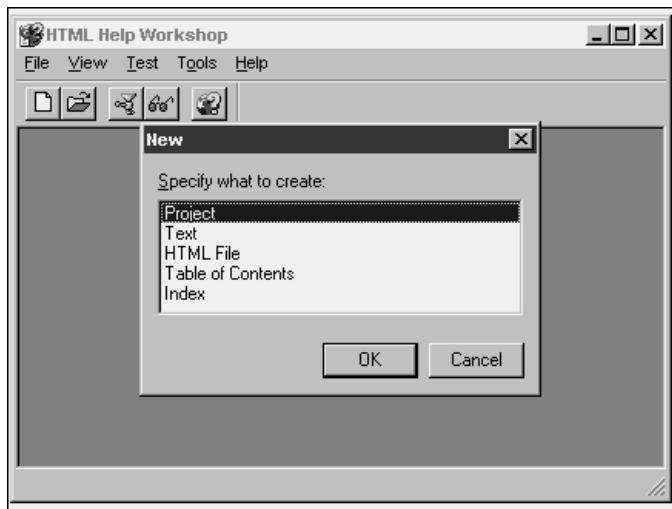
But eventually you'll have a series of HTML files (and associated image files). Pop those into your Help directory for your application, or wherever you want to work with them, and continue on.



**Figure 25.6.** Before creating a new HTML Help project, create a directory in your application directory structure for it.

## Creating your HTML Help project

Finally, now, you can run the HTML Help Workshop program—off the Start menu unless you've already gotten ahead of class and created a desktop shortcut. Select the File, New menu option. You can choose several different types of files to create, but first you'll want to pick a Project, as shown in **Figure 25.7**.



**Figure 25.7.** Create a new Project in HTML Help Workshop.

You'll then be asked to name your HTML Help project and identify where it's supposed to be created, as shown in **Figure 25.8**. This is a fairly lame interface, because it doesn't use standard controls to ask you to perform the task. If you just click the Next button, you'll end up creating an HTML Help project in the Program Files\HTML Help Workshop directory on drive C or wherever you installed it. You don't really want to do this, do you? Instead, you should point to the Help directory under your project directory.

Instead, click the Browse button to open the File, Open dialog, and navigate to the desired directory. Then enter the name you want to use for your project file (yes, even though the dialog says "Open") and click OK. The path and project file name will be filled in for you in the wizard, as shown in **Figure 25.9**.

You'll be asked a few more questions—which you can ignore for the time being—and after you click the Finish button, you'll get an empty project as shown in **Figure 25.10**.

Now that you've got a project, you're going to want to start organizing the hierarchy of topics. Click the Contents tab.

You'll be warned that you don't have a Table of Contents file yet, as shown in **Figure 25.11**. Select "Create a new contents file" and click OK.

You'll then be asked for the name of the Table of Contents file (with an .HHC extension). I always accept the default "Table of Contents.HHC" name.

You'll then see the empty Contents tab, as shown in **Figure 25.12**.

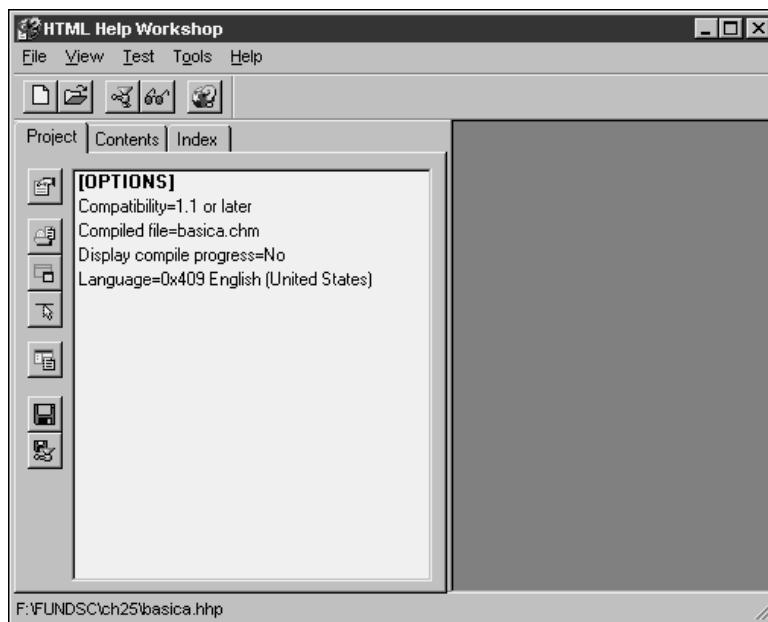
You'll want to first add sections and levels, and then add topics to those sections and levels. I'm going to build a simple help file with two sections, add a sublevel to the first section, and then add two topics to the sublevel.



**Figure 25.8.** Don't simply enter a name for your project and click Next; you'll be creating a project deep in the bowels of the Program Files directory—probably not where you want to.



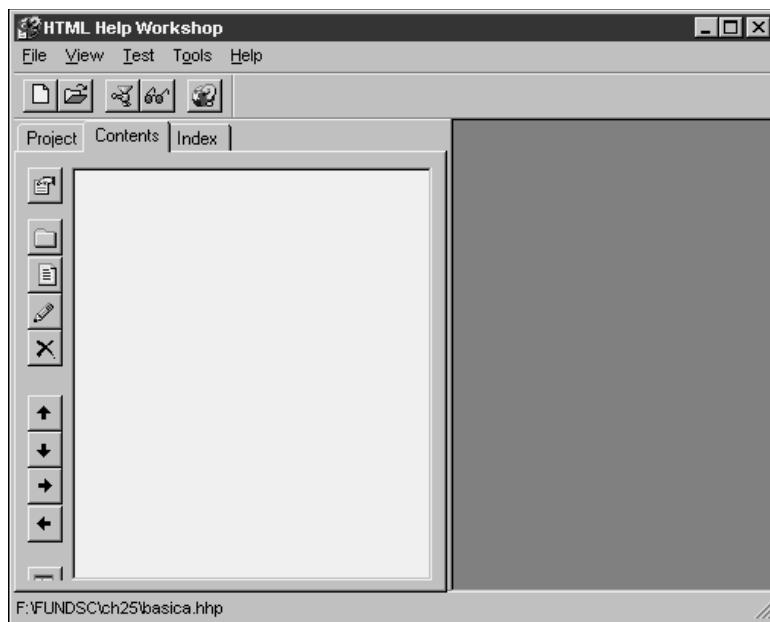
**Figure 25.9.** Using the Browse button, you can select the directory and enter a file name for your project.



*Figure 25.10. An empty HTML Help project.*



*Figure 25.11. Choose to create a new contents file when starting a new help project.*



**Figure 25.12.** The empty Contents tab.

### Create levels for your help file

There are 11 buttons down the left side of the Contents tab (including one that is partially hidden and another that is completely hidden) in Figure 25.12. These are, in order:

- Contents properties
- Insert a heading
- Insert a page
- Edit selection
- Delete selection
- Move selection up
- Move selection down
- Move selection right
- Move selection left
- View HTML source
- Save file

It should be obvious what these do, now that you know their names.

To create levels for your help file, you should first add headers for each section. In this example, the two headers are User Guide and Reference Guide. Click the Insert a heading button to open the Table of Contents Entry dialog as shown in **Figure 25.13**.

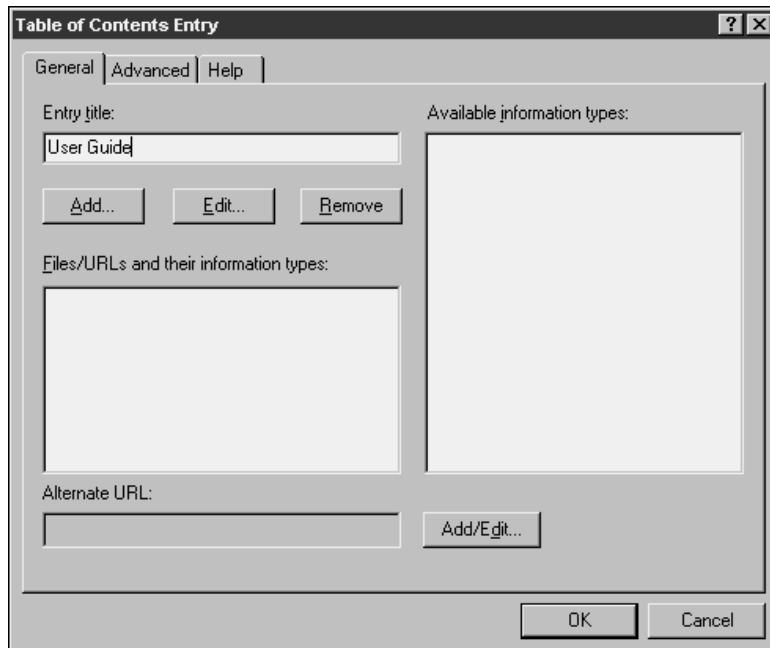
 You might be wondering why the button says “Insert a heading” but the title bar of the dialog says “Table of Contents Entry.” You’ll find inconsistencies like this throughout the HTML Help Workshop tool. While wonderfully powerful and generally easy to use, HHW is a very immature product, and has a lot of “fit and finish” issues that still need to be fixed.

This dialog is sort of confusing, because it can do a *lot* of things. For example, if you want to have an HTML page associated with the User Guide section heading, you can click the Add button to grab an existing HTML page on disk.

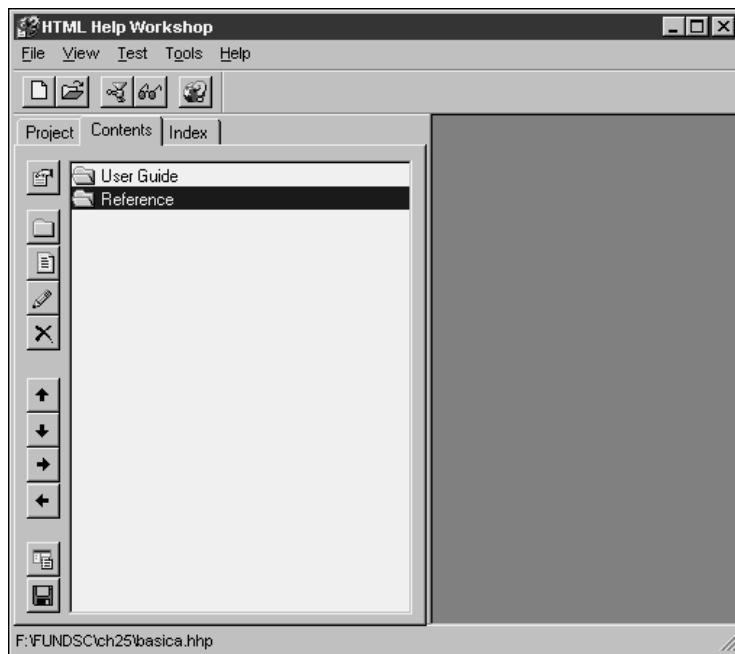
Right now, though, I’m going to take the easy way out and just enter a new entry title, as shown in Figure 25.13, and then click OK.

Repeating these steps for a Reference section heading produces a Table of Contents tab that looks like **Figure 25.14**.

Next, it’s time to add topics.



**Figure 25.13.** Adding a Table of Contents entry.

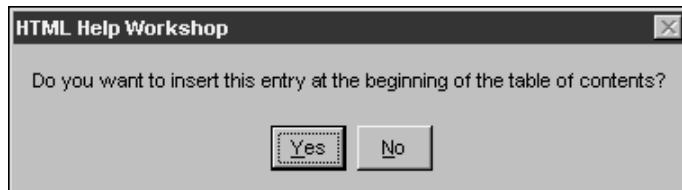


**Figure 25.14.** The Contents tab with two entries.

### Add topics underneath a section heading

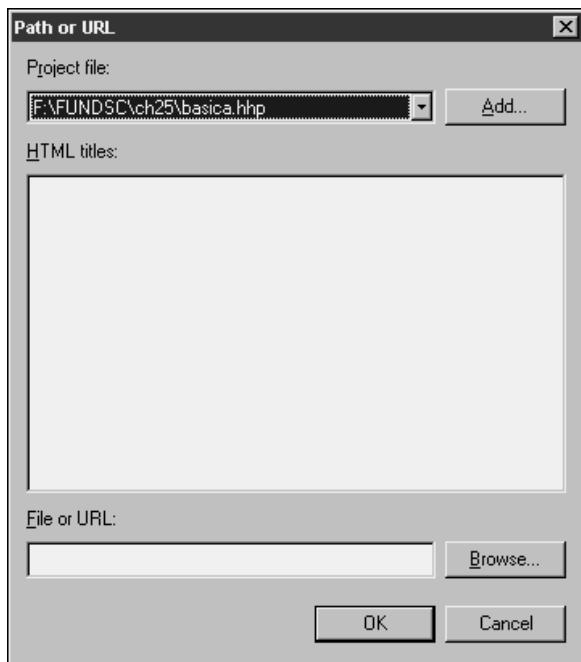
Select the User Guide section heading, and click the Insert a page heading button. You will be asked whether you want to add this entry at the beginning of the Table of Contents, as shown in **Figure 25.15**. If you answer yes, the page you add will be placed in the Table of Contents above the User Guide section heading. This is how I created the Introduction title page for the TWISTOR help file shown in Figure 25.1.

Answer No, and the Table of Contents Entry dialog in Figure 25.13 will appear again. This time, however, it's time to add a real page—not just a section heading. In fact, if you just type an entry title and then try to click OK, you'll be warned that you must associate a file or a URL with the entry.



**Figure 25.15.** You'll be warned if you try to insert a page before the first section heading.

The next couple of steps get tricky. You'll click the Add button immediately below the Entry title text box, and the Path or URL dialog will appear as shown in **Figure 25.16**.



**Figure 25.16.** The next step to adding a topic page is the Path or URL dialog.

You can manually enter a file or URL in the text box at the bottom of the dialog, or click the Browse button to open yet another Open dialog as shown in **Figure 25.17**. Tired of cascading dialogs yet?

Select the file you want, click the Open button, and then click the OK button to get back to the Table of Contents Entry dialog. You'll want to type an Entry title if you haven't already, as shown in **Figure 25.18**, and then click OK. I named this entry "Quick Start."

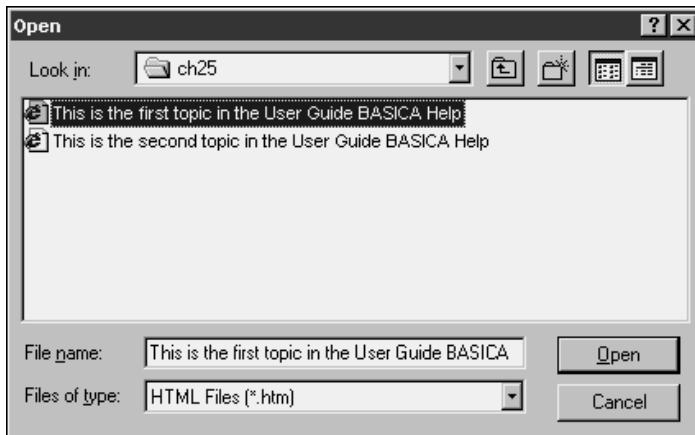
When you close the Table of Contents Entry dialog, you'll see your page added to the Contents tab. You can use the four arrow buttons to adjust the position of the page if it landed in the wrong place, or if you want to move it somewhere else. See **Figure 25.19**.

You can have a grand time adding and rearranging section headings and topic pages. Save your work every so often, as you would in any other tool. Once you're done, you can build your .CHM file to see what it looks like, or you can add index and search capabilities.

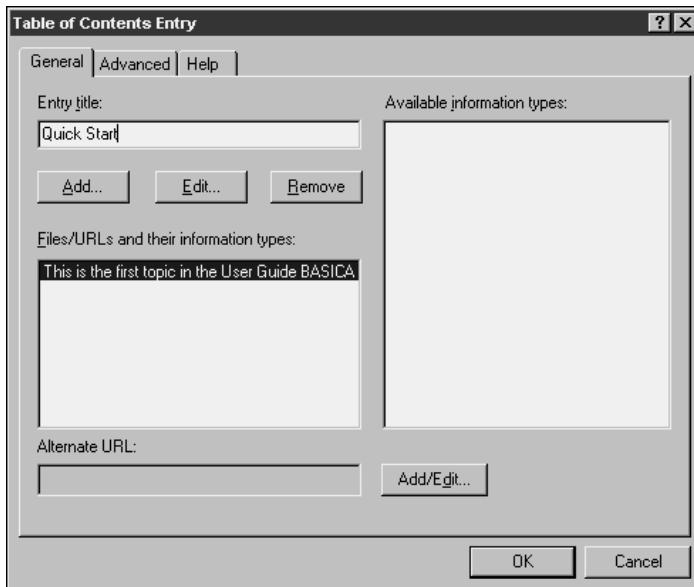
## Setting .CHM file options and building the .CHM file

Building your .CHM file is fairly trivial, but you'll probably want to set a few options first. You can set some options from the Project tab and others from the Table of Contents tab.

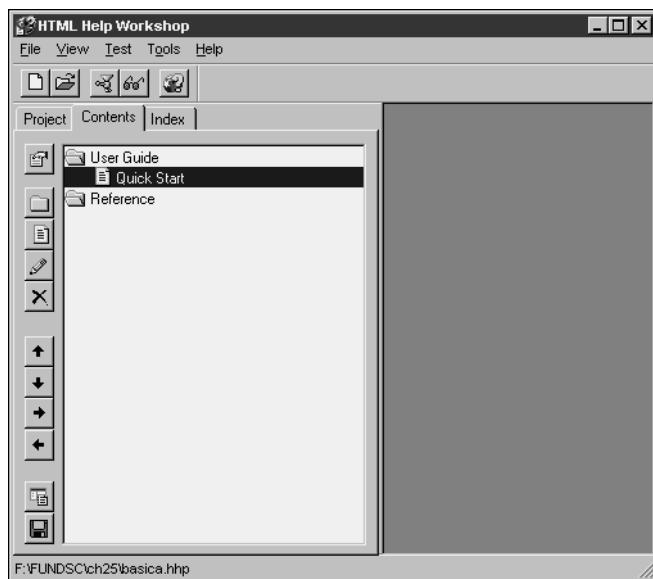
The first option to set is the string that is displayed in the title bar of your .CHM file. Select the Project tab and click the Options button to open the Options dialog as shown in **Figure 25.20**. Type the title in the Title text box.



**Figure 25.17.** Click the Browse button in the Path or URL dialog to display the Open dialog, from which you can choose the file you're looking for.



**Figure 25.18.** Before you close the Table of Contents Entry dialog, be sure to type the text you want to display in the Table of Contents in the Entry title text box.



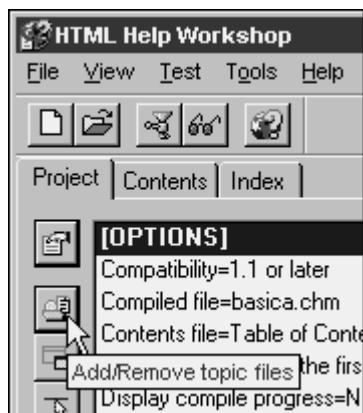
**Figure 25.19.** The Quick Start page after being added to the project.



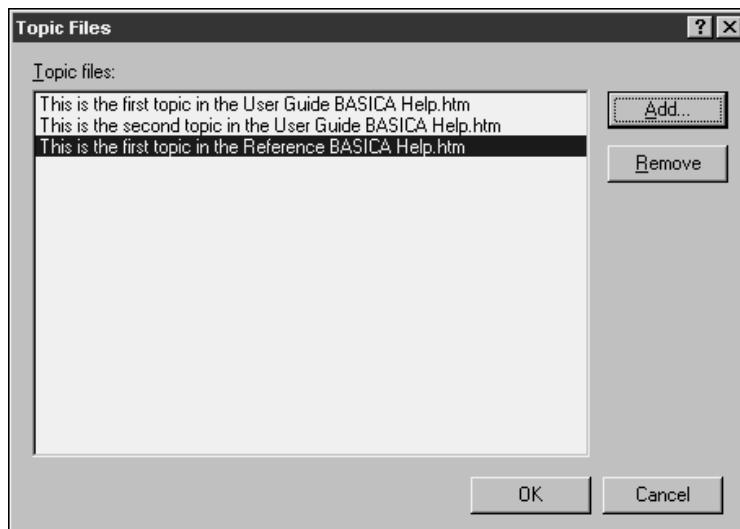
**Figure 25.20.** Type a title in the project's Options dialog.

The next option to set is the default page—the topic page that displays when the .CHM file is first opened. To do this, you'll first need to add the topic file to the help project, just as you do with VFP projects. To do so, select the Project tab and click the Add/Remove topic files button (the second from the top) as shown in **Figure 25.21**.

You'll get the Topic Files dialog as shown in **Figure 25.22**. Click the Add button to add one or more topic files to the list, and then click OK. Why add more than one? I'll explain in the section on indexing.



**Figure 25.21.** Click the Add/Remove topic files button in the Project tab to open the Topic Files dialog.



**Figure 25.22.** Use the Topic Files dialog to add topic files to your help project.

Once you've added the topic files to the project, the Default file combo box in the General tab of the Options dialog will be populated with those files. Select the topic file you want as your default file, as shown in **Figure 25.23**.

Another option you might want to select right away is on the Compiler tab of the Options dialog. Select the Progress check box in the "While compiling, display" box as shown in **Figure 25.24** so that you can see what's happening. In a small project, you won't really notice or care, but in a project with 100 or more files, you might end up waiting for a while, and you'll want to know that the compilation is running—not locked up.

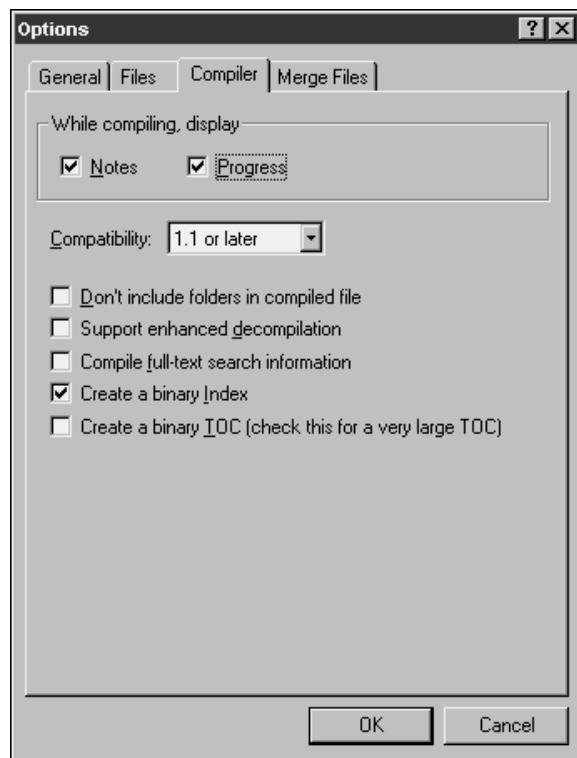
When you're done selecting options, click the Compile button (there are several of them—including one on the top toolbar and another on the bottom of the list of buttons down the left side of the Project tab). You'll be greeted with the dialog shown in **Figure 25.25**.

In the build I'm using, checking the Automatically display compiled help file check box has no effect—I still have to run the .CHM file manually by double-clicking it in Windows Explorer. Nonetheless, doing so produces the simple .CHM file as shown in **Figure 25.26**. A lot easier than the rigmarole necessary for WinHelp.

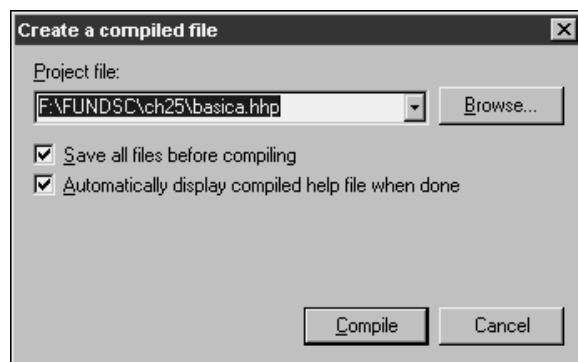
But you want more, right? Always more. What about those fancy Index and Search tabs?



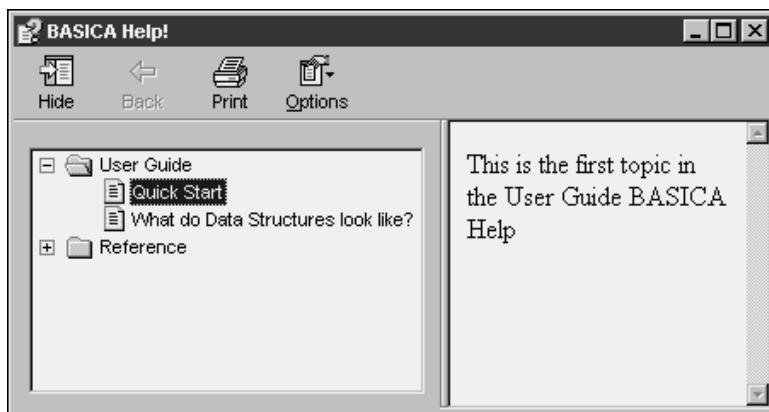
**Figure 25.23.** Select the topic file you want as the default page in the General tab of the Options dialog.



**Figure 25.24.** Select the Progress check box in the Options dialog.



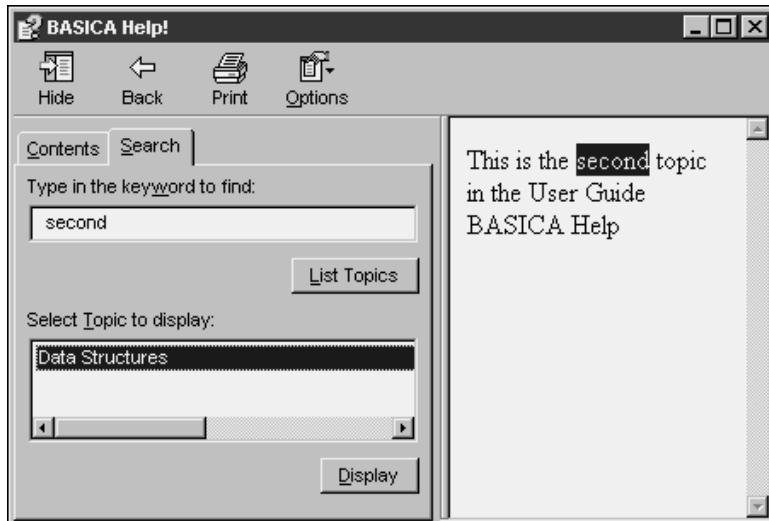
**Figure 25.25.** The Create a compiled file dialog.



**Figure 25.26.** The completed HTML Help .CHM file.

### Add index and search capability

Adding search capability is trivial, actually. Look back to Figure 25.24 and notice the Compile full-text search information check box about halfway down the dialog. Checking this box and recompiling will automatically provide a Search tab as shown in **Figure 25.27**.



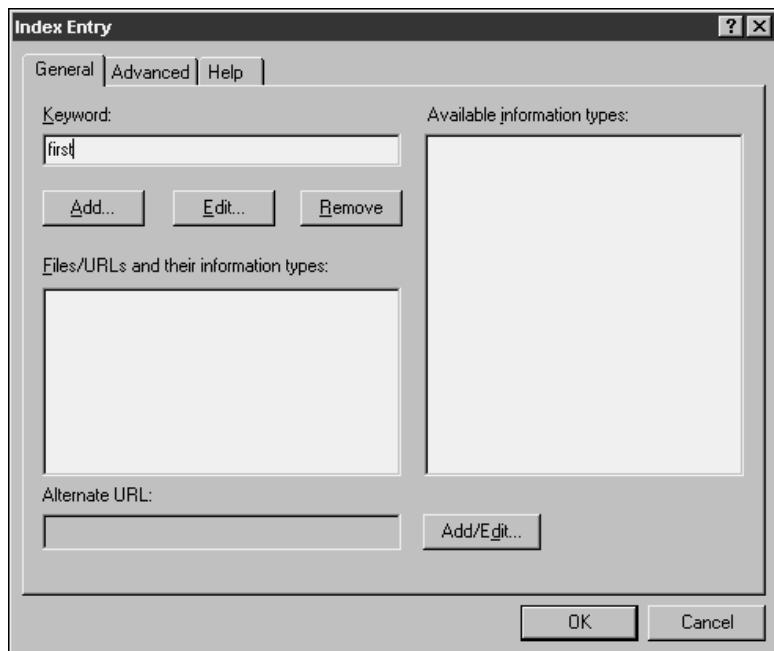
**Figure 25.27.** The completed HTML Help .CHM file with Contents and Search tabs.

Adding an index is considerably more complex, comparatively.

First, you need to create an index file with keywords mapped to specific topic pages. Then you have to add the index file to the help project. Third, you need to recompile your help project.

Suppose you wanted to create keywords for “first” and “second,” and map the first User Guide and Reference pages to the “first” keyword, and the second User Guide page to the “second” keyword.

First, select the Index tab in the HHW. Next, click the Add a keyword button (the one with the key) to open the Index Entry dialog, as shown in **Figure 25.28**. Add the keyword you want, such as “first.”



**Figure 25.28.** Adding a keyword using the Index Entry dialog.

Next, click the Add button to open our old friend, the Path or URL dialog, as shown in **Figure 25.29**. You’ll notice a couple of odd things in this dialog. First, you don’t see file names in the HTML titles list box—instead, there are phrases of some sort. Where did these come from?

When you create a Word document, you can edit the properties for the document by selecting the File, Properties menu option. When you do so, you get the dialog shown in **Figure 25.30**.

The title of the Word document gets translated to the HTML file’s <title> tag. If you didn’t do so, you could manually edit the HTML file and replace the <title> tag yourself, but that

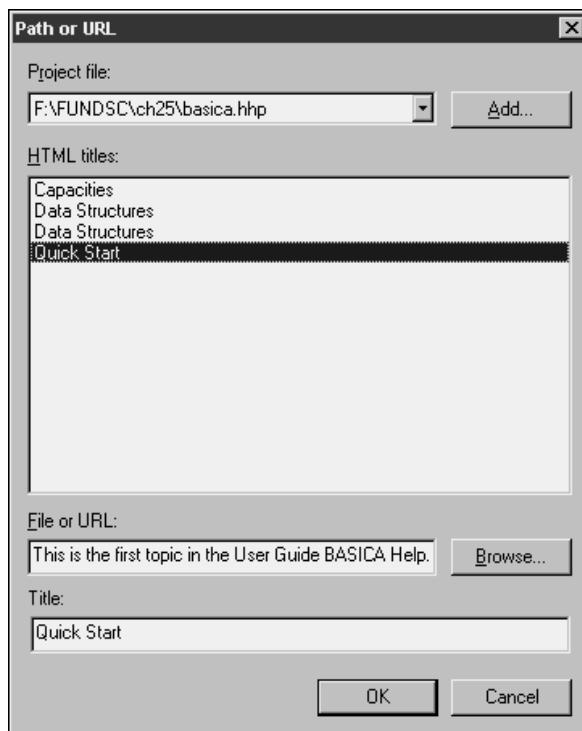
would be a pain, because if you changed the Word file and then resaved to HTML, you'd lose your change to the <title> tag, right?

This <title> tag in the HTML file is where HHW gets the descriptions in the HTML titles list box. This has nothing to do with the Table of Contents entries, so I usually make the title the same name as the Word document file—it's easier to identify the file later on.

The second question you might have relates to the two “Data Structures” entries in the HTML titles list box. There are a couple of reasons you might run into this. The first is that you copied a Word document, and forgot to change the title of the copy. This happens to me all the time. You can determine which title belongs to which document by looking at the File or URL text box while the HTML title is highlighted.

The other thing that might have happened is that you added a HTML file to the project, and then removed it later for one reason or another. HHW is not very good about cleaning up after itself when you delete files—and here's one example. I added a file (which had a title of “Data Structures”) and then deleted it. Unfortunately, the file list still thinks it's there, so it displays two copies. I hope this will be fixed in a future version, because it means that mistakes on your part when you delete files can be very confusing.

Let's suppose, however, that we can overlook this minor transgression. You can add more than one topic to a keyword, ending up with an Index Entry dialog like in **Figure 25.31**.



**Figure 25.29.** The Path or URL dialog showing files from the project file list.

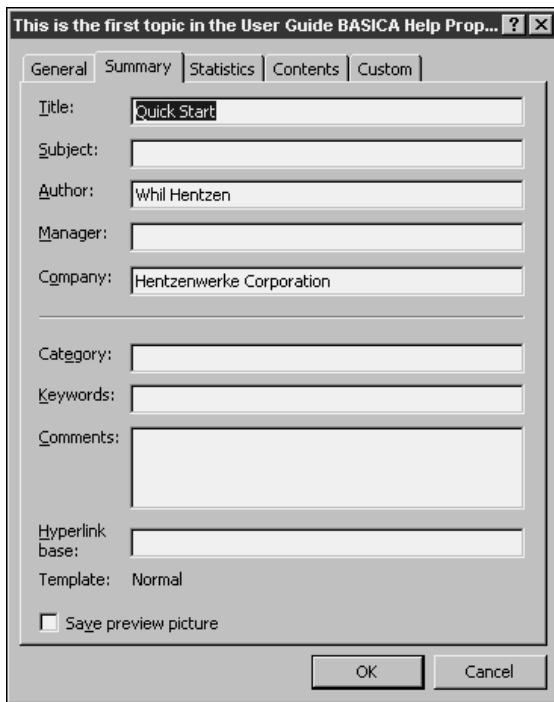


Figure 25.30. The Properties dialog for a Word 2000 document.

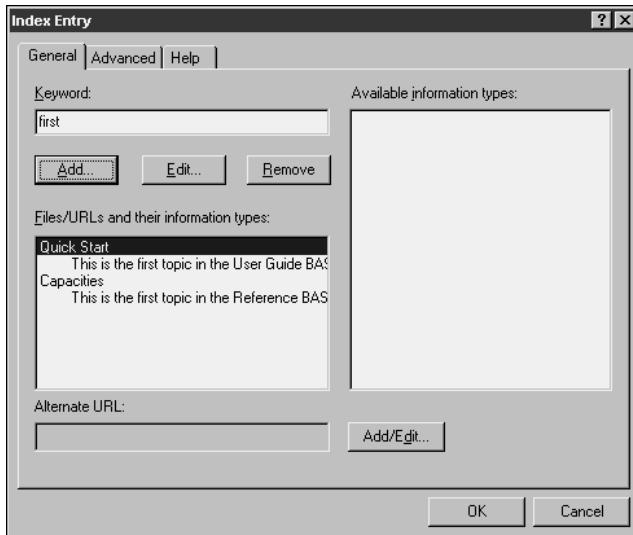
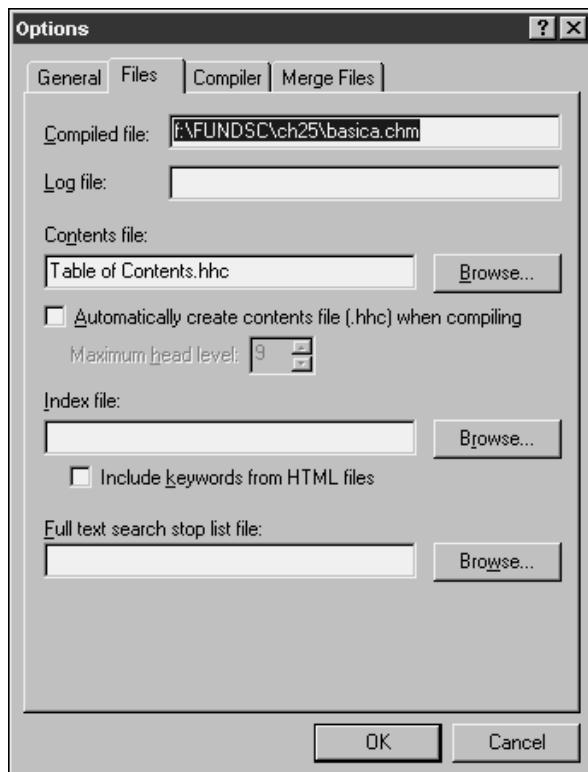


Figure 25.31. An index keyword with two topics associated with it.

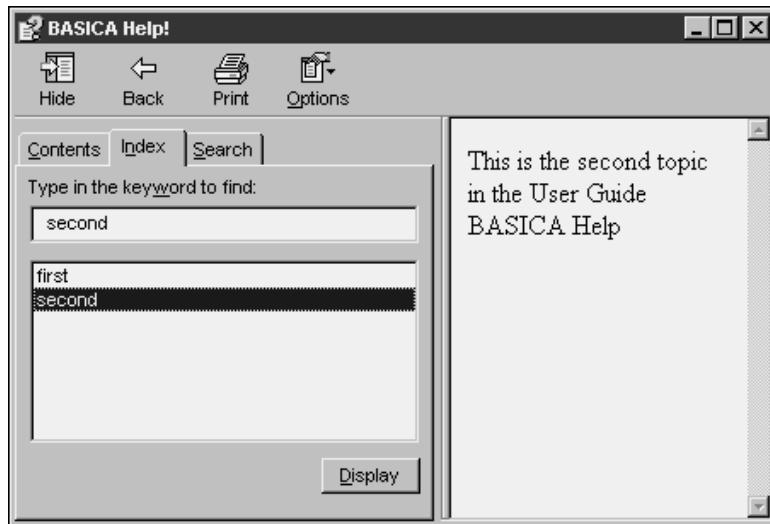
Continue to add keywords to your index list until you're done, and ready to compile again. However, you still need to tell HHW to use that index information. Pop over to the Files tab of the Options dialog as shown in **Figure 25.32**, select the Index file (it's got an .HHK extension, and I always keep mine named "Index"), and check the Include keywords from HTML files check box. Then save and compile away!

When you compile and run your .CHM file, you'll get an Index tab in between the Contents and Search tabs, as shown in **Figure 25.33**.

 Now that you're acquainted with the basic HTML Help mechanism, you'll probably want to play around with it a bit. If you get serious about building HTML Help for your applications, you'll probably want to use a third-party tool—I would recommend Rick Strahl's HTML Help Builder, described in Chapter 16. Go back and check it out right now! There's an excellent tutorial for HTML Help Builder by Doug Hennig in the December 1999 issue of FoxTalk ([www.pinpub.com/foxtalk](http://www.pinpub.com/foxtalk)).



**Figure 25.32.** Use the options on the Files tab of the Options dialog to include an index file in your HTML Help .CHM file.



**Figure 25.33.** The completed HTML Help .CHM file with an *Index* tab.

## Including a .CHM file with your VFP application

There are three things you might want to do. The first is to have your VFP application call your .CHM file. Sounds reasonable, doesn't it? The next thing you're probably going to want is context-sensitive help—so when a user clicks "Help" while a form is open, a help topic for that form is displayed. And the third thing is to include your .CHM file and the HTML Help rendering engine with your distribution files.

### Calling a .CHM file from your VFP application

This is pretty easy—see for yourself by simply grabbing the .CHM file for this book (F.CHM) and putting it in your VFP path. Then, in the Command window, issue the commands:

```
set help to F.CHM  
help
```

and the Fundamentals HTML Help file will appear. These two commands are all you need in your application as well. After you include them in, say, your startup or top-level program, pressing F1 or selecting the Help menu option (assuming, of course, that your Help menu option runs the HELP command) will bring up the HTML online help and display the default topic page.

### Creating and calling context-sensitive help topics

The next thing you're going to want to do is include context-sensitive help in your application. In other words, clicking on a Help button somewhere on a form will bring up a help topic specific to that form instead of the default topic page in your .CHM file.

---

There are three steps to this process:

1. Identify your help file, as I showed in the previous section:

```
set help to F.CHM
```

2. Call a specific topic in the .CHM file from the form. First, I put a Help button on every form in my application (via the Form class, of course). Next, I have a Help() method on that form (also courtesy of the Form class). The Click() method in the Help button calls the Help() method:

```
click()  
thisform.help()
```

The Help() method—in the Form class—contains the following code:

```
set topic to alltrim(thisform.caption)  
help  
set topic to
```

Thus, when the user clicks the Help button, the help topic is set to the caption and then the .CHM file is opened.

3. Make sure there's something to call—this is the most time-consuming and detail-oriented part. When you create a help topic for a specific form, you need to make sure that the caption for the form (for example, "Vendor Maintenance") matches the Entry title in the Table of Contents Entry dialog in the Contents tab of HHW for the topic page. Yes, it's just that easy. No fancy ID values or separate context ID mapping files like you had to do in the olden days—just make sure your Entry title matches the caption!

I always have someone else run through the whole application, testing every Help button to make sure I haven't misspelled something or another.

## Distributing HTML Help and your .CHM file with your VFP application

Distributing your .CHM file is no big deal—just include it as part of the files in your application. I put the help file in the APPFILES directory because it should be available to the application regardless of which data set is being used.

You'll also have to make sure that the HTML Help rendering engine is included with your distribution files, which can be a real pain. The easiest way is to have your users separately install Internet Explorer 4.1 or better—this has been the best way I've found to make sure that the correct versions of all rendering engine files are present. Of course, there are a lot of people who don't want to do this, so you can also check the HTML Help check box in the Visual FoxPro Setup Wizard, and as of Service Pack 3, I think it's working, but I've had trouble with it before and haven't tested it in SP3.



# Chapter 26

## Distributing Your Applications With the VFP Setup Wizard

An application isn't much good if it never gets to the users. In the olden days, distribution was the easy part—you copied a bunch of files from your distribution machine onto a floppy disk, and then copied those same files onto your user's PC. It's considerably more complicated these days, and Visual FoxPro has a tool—the Setup Wizard—that allows you to package and distribute your application, much like VFP's installation routine itself works. In this chapter, I'll show you how to prepare your application for the Setup Wizard, walk you through the Setup Wizard, and point out a few options to be aware of, in case you need to take advantage of them yourself.

One of the nice things about Windows is the standardization of tasks and processes that ought to be standardized. Many of you have suffered through multiple incarnations of DOS and the applications that ran on top of it—and each application required you to perform a wide variety of different steps in order to accomplish common tasks such as printing and saving files. The initial installation of the application was another common task that could be done in many different ways.

With well-written Windows applications, however, the installation routine is always the same. Your application comes packaged as a set of files, including a SETUP.EXE installation routine, and one or more .CAB (“cabinet”) files that contain all your application files.

Upon running SETUP.EXE, you are required to respond to a license dialog, then prompted for the location and name of the directory where the app is to be installed, and perhaps a few other questions specific to the application you are installing.

After installation is complete, the routine has created a new program group and perhaps a set of menu items under it, as well as an icon on the desktop. The installation program might have even offered to launch a Readme file.

With Visual Studio 6.0 and Visual FoxPro 6.0, you can distribute your applications to your end users with the same type of installation routine. You don't have to buy third-party programs or spend extra money to create a set of distribution files that contain an installation routine and all of the files—data and programs—that make up your application.

The tool you use with Visual FoxPro 6.0 is the Setup Wizard—yes, a *wizard*. For those of you who feel that Real Programmers Don't Use Wizards, you're wrong on this point. The Setup Wizard walks you through every step you need to cover to create a set of files that you can distribute to your users, and it's far too much work to install a VFP application otherwise.

Once your user has gotten the files—either via a network share, a CD, or downloaded from the Web—they'll run the SETUP executable, answer a couple of questions, and the custom app you developed for them will be installed and ready to rock and roll.

But there are a couple of things you're going to want to know before running the Setup Wizard, so I'll start at the beginning.

## What does the Setup Wizard do?

To use the Setup Wizard, you need to have a copy of your application (in a separate directory) as it will appear on your user's machine. Generally, that means you'll compile your application to a standalone .EXE, and then copy this .EXE along with any other necessary files to a new directory, and run the Setup Wizard on that group of files. (If you're creating a middle-tier component, you'd actually be compiling to a .DLL, but the idea is pretty much the same.)

What other necessary files might there be? Most applications include, at the very least, an empty copy of the database to be used with the application. If yours does, that will be one set of files. You might also include some "system" files, like a data dictionary, a table that stores next available primary keys, a table of users, a help file, a couple tables of standard lookup codes, and so on. These all go on the pile as well.

There's one other set of critical files that are needed—but that you don't have to worry about. The standalone .EXE file (or .DLL) that you create just contains your code—and while your application is running, it needs to have access to the Visual FoxPro engine in order to work. However, you don't have to include a copy of VFP with your application, nor do you have to require your users to buy VFP. Instead, Visual FoxPro comes with a pair of "runtimes" that include all of the necessary functionality to provide support for your own .EXE-based or .DLL-based application. One runtime is for .EXEs and the other is for .DLLs. And, even better, you don't have to copy this Visual FoxPro runtime to this directory—I'll explain why in just a second.

Once you've gotten all your files together, you'll run the Setup Wizard, make a number of choices (which I'll walk you through in just a couple of pages), and click the Finish button. The Setup Wizard will crank for a bit (depending, of course, on the size of your application and the horsepower of your machine), and finish up with a group of files that you'll distribute to your users.

Because you're running the VFP Setup Wizard, it knows where to find the VFP runtime files and just grabs a copy while it's creating the distribution files. You don't have to do anything extra here.

The files that you distribute to your users can be packaged in one of several ways, depending on one of the choices you make in the Wizard, but the process the user follows will essentially be the same.

First, they'll run the SETUP executable that was created by the Setup Wizard, they'll load in floppy disks or CDs (or point to files on a network share) as appropriate, and then go through a standard Windows install. If they've installed Word or Excel or most any other common Windows program, they'll be right at home. A few minutes later, they'll be all set to run your application.

Behind the scenes, of course, the installation routine is doing more than would initially appear. For instance, remember that Visual FoxPro runtime that the Setup Wizard grabbed and automatically included in the distribution files? The installation routine automatically puts that file in the Windows SYSTEM (or SYSTEM32) directory—you won't find a copy of it in the root directory of your application.

The installation routine also automatically creates a series of registry keys for the application, which are used for rerunning setup and uninstalling. And, of course, the routine adds an entry in Add/Remove Programs, so that your users can remove the application just like

every other well-behaved Windows app. The installation routine will even install and register ActiveX controls and the HTML Help engine.

## Are you running SP3?

The first thing you have to do to prepare for distribution has nothing to do with programming, testing, or anything as far as your application goes. You want to make sure you are running Visual FoxPro Service Pack 3.

You can tell if you are by selecting the Help, About menu option and examining the first line in the About dialog—it should say “Microsoft Visual FoxPro 6.0 SP3”, and if you use the VERSION() function, you’ll get

```
Visual FoxPro 06.00.8492.00 for Windows
```

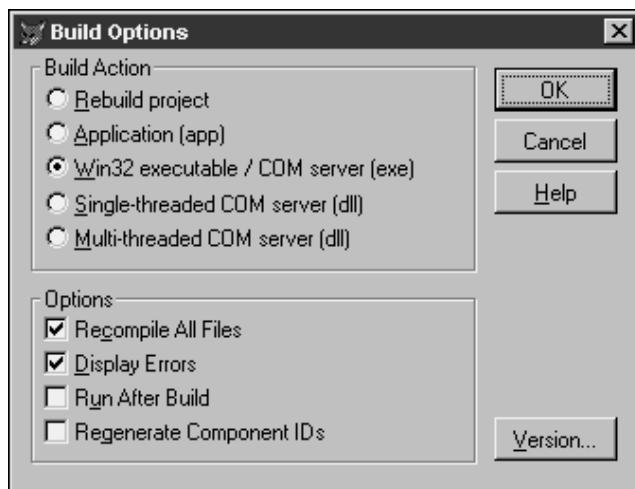
(The original version of VFP 6.0, for you trivia buffs, was 8167, by the way.)

The original version of the VFP 6.0 Setup Wizard has a number of bugs that can cause problems if you make heavy use of the various options, or if you depend on a couple particular configurations, such as multiple directories with files of the same names in both directories. In between the time that VFP 6.0 was released and SP3 was released, a new Setup Wizard that fixed these issues was made available on the Microsoft Web site, and that Setup Wizard has been incorporated into SP3.

## Preparing your application

Now that you have updated VFP properly, you’ve got to get your house in order.

First is to build an executable. When you click the Build button in the Project Manager, you’ll get the Build Options dialog as shown in **Figure 26.1**.



**Figure 26.1.** Select the type of file to build when preparing for distribution.

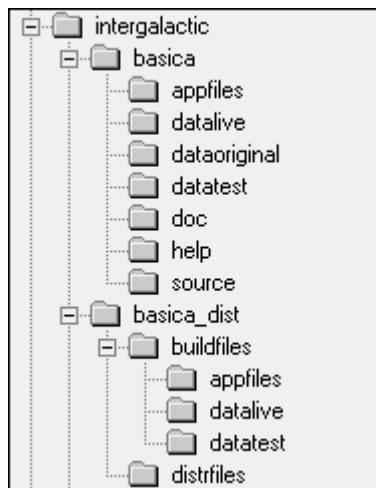
If you have a dialog that looks different—in other words, just four option buttons in the Build Action box—it means you’re not running Service Pack 3. SP3 also gives you the ability to create three types of COM servers, while earlier versions gave you only two.

How do you decide which type of Build Action to choose? First of all, when you are building your application in order to distribute it, you’ll need to choose one of the last three. You can choose Application (app) if you’ve already created an umbrella application that is started by an .EXE file, and have that .EXE call the .APP file. However, in this case, you won’t be using the Setup Wizard to distribute the .APP file by itself.

If you’re creating a LAN or client-server application and you’re using Visual FoxPro as the front end—the user interface—select the Win32 executable/COM server (exe) option.

If you’re creating a COM server that will be used as a middle-tier component, you’ll select either Single-threaded or Multi-threaded COM server, and still use the Setup Wizard for packaging the necessary files and creating distribution disks. Just because a middle-tier app doesn’t have a UI doesn’t mean you can get away without the Setup Wizard. Your middle-tier app is still a VFP app—and, as such, still needs a Visual FoxPro runtime. It’s just a matter of which runtime will be needed. (The Setup Wizard is smart enough to know which one to grab.)

The second step is to make a directory that contains the application files, as you want them to appear on your user’s machine, and a directory where the resulting files will go. In Chapter 16, I mentioned that I use a pair of directories as shown in **Figure 26.2**.



**Figure 26.2.** The directory structure for your distribution files.

The application’s development directory is called BASICA in Figure 26.2. The BASICA\_DIST directory is used for all the various bits and pieces of packaging and distribution. BUILDFILES is the new directory I was talking about in terms of containing all the application files that will appear on your user’s machine. As you see, the BUILDFILES directory in Figure 26.2 contains two data directories and an APPFILES directory that contains system files and whatnot.

The application's .EXE file goes in the root under BUILDFILES, along with a CONFIG.FPW file and any other files you'll want in your application's root directory. For example, I include a README.TXT file and an icon file for the application.

When you run the Setup Wizard, you'll be asked where you want to place the files that you're going to distribute to your users. This is what the DISTRFILES directory in Figure 26.2 is for. You'll want to create this directory before you run the Setup Wizard—I've run into occasional problems if I try to switch over to Windows Explorer and try to create it while I'm between steps 5 and 6 in the Setup Wizard dialog.

Another directory you might want to have available under BASICA\_DIST is called ZIPS, although it's not shown in Figure 26.2. I find I like to have copies of the previous distribution files I've sent out—and each of those copies goes into this directory.

Now it's time to run the wizard!

## Running the wizard

You can find the Setup Wizard in the same submenu as the rest of the Visual FoxPro wizards—under the Tools, Wizards menu option.

I'll discuss each screen of the Setup Wizard in this section, and then I'll cover some of the options, such as handling ActiveX controls and which distribution options to choose.

By the way, all of the choices you are about to make will be stored in a file called WZSETUP.INI in the VFP HOME() directory. This file will be used to populate the controls of the Setup Wizard when you run it next, on the assumption that you're probably going to rebuild the last application you built. The first time you run the Setup Wizard, it will warn you and ask you if you want to create it. Well, duh...

### Step 1: Locate files

The first screen asks you to point to the root directory of the copy of the application from which you want to build distribution files, as shown in **Figure 26.3**.

As you can see, I pointed to the BUILDFILES directory under the BASICA\_DIST directory.

### Step 2: Specify components

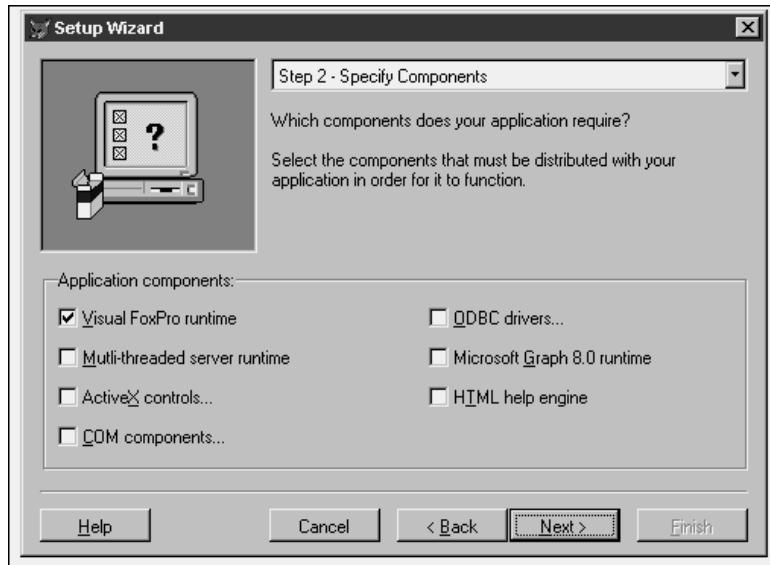
The second step is to identify which extra goodies you want to include in your distribution files, as shown in **Figure 26.4**.

#### Runtimes

The first thing you'll need to do is identify which of the Visual FoxPro runtimes you want to ship with your application. You have the choice of either the Visual FoxPro runtime or the multi-threaded server runtime. Basically, if you're building multi-threaded COM servers, you'll want to include the multi-threaded server runtime. For all other applications, including client/server front ends, out-of-process COM servers (.EXEs) and single-threaded COM servers (.DLLs), you'll want to use the regular VFP runtime.



**Figure 26.3.** Point to the root directory of the copy of the application from which you want to build distribution files.



**Figure 26.4.** Step 2 allows you to specify which components you want to include as part of your application.

### ActiveX controls

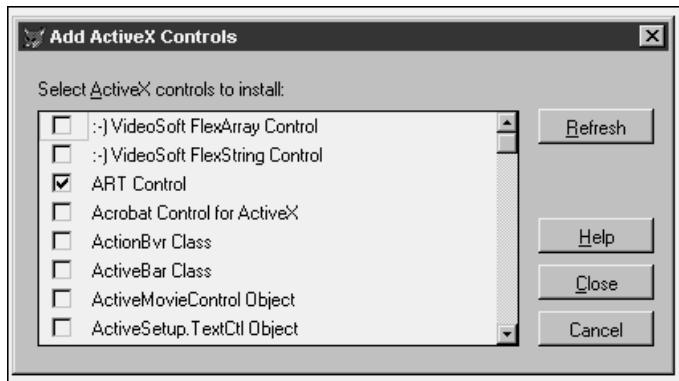
The next check box, ActiveX controls, is yet another example of a stupid user interface. As I've bitched about before, a check box is a control to indicate status of something, but not to initiate an action. However, this one also performs an action—well, sometimes it does. If the check box is unchecked, checking it will open a form similar to the one shown in **Figure 26.5**. You then use this Add ActiveX Controls form to select (or deselect) ActiveX controls to install.

However, once you have selected at least one control and then closed this dialog, the ActiveX controls check box in the Step 3 dialog of the Wizard will be checked.

What's so stupid about this? What if you want to add another ActiveX control? The check box is already checked—clicking on it again simply clears the check box. You have to click it a second time to check the box again and open the Add ActiveX Controls dialog in order to do more work on it. This interface should really have had a command button and another control indicating that one or more ActiveX controls were selected. Anyway, enough whining.

All of the ActiveX controls that are registered with Windows should show up in the dialog—in fact, when you open this dialog, the Setup Wizard will search through the Windows Registry to determine what's available at the current time.

Note that you must do this step to explicitly identify which controls have to be added.



**Figure 26.5.** Use the Add ActiveX Controls dialog to identify controls to include with your distribution files.

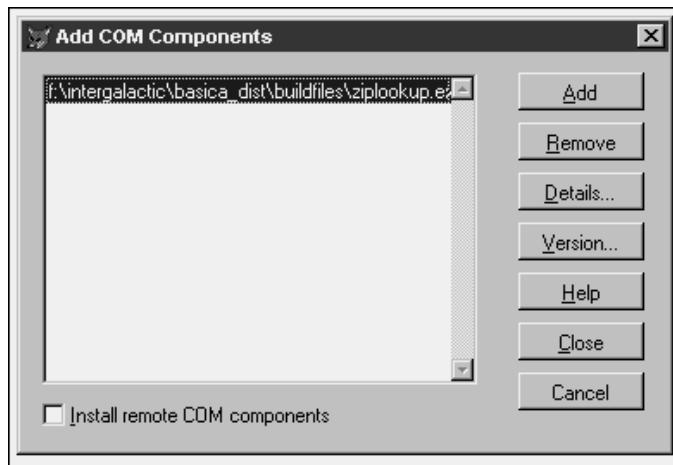
### COM components

If you have COM components—either .EXEs or .DLLs—you specify that you want to have their files included in your distribution files by selecting the COM components check box.

When you initially check the box, the dialog shown in **Figure 26.6** will open, and the list box will be empty. You'll click the Add command button to add files to the list box. When you close the dialog, the COM Components check box on the Setup Wizard dialog will be checked.

I won't belabor the point about how stupid this check box's interface is—it works pretty much the same as the ActiveX controls check box. The only difference is what is displayed in the Add COM Components list box. When you uncheck the check box, the box will remain unchecked—but when you check it again, any files that you had added to the list box will still

be there. Awkward to get used to, but you probably won't be in this dialog a lot, so it shouldn't cause you a lot of stress.



**Figure 26.6.** Use the Add COM Components dialog to identify components to include with your distribution files and specify options for those components.

When you click the Add button in the Add COM Components dialog, you'll be able to select either .EXE or .DLL files. If you select a .DLL, it is by definition a local server—one that is running on the user's local machine. If you select an .EXE, however, you can choose to install and run that server on another machine. To do so, click the Details command button in the Add COM Components dialog and open the COM Component Details dialog, as shown in **Figure 26.7**. This dialog also allows you to identify which machine to use as a target.

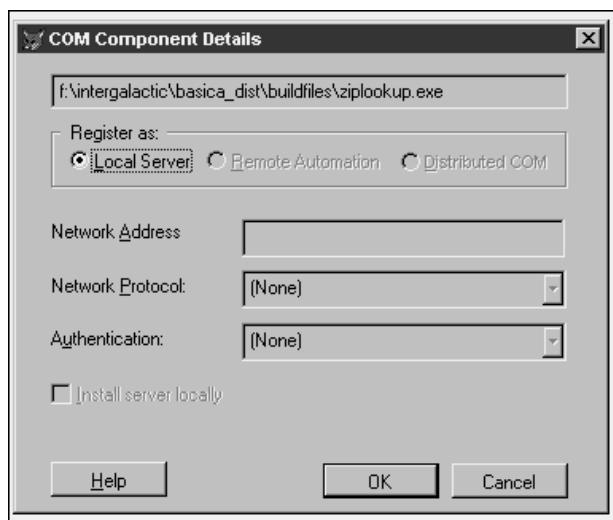
You'll need to know the Network Address of the target machine, and then you can choose which Network Protocol and Authentication method to use. You can leave the Network Address, Network Protocol, and Authentication fields empty if you want the user to enter these values during installation.

When you get back to the Add COM Components dialog, you'll want to check the Install remote COM components check box so that the installation routine will automatically register your remote servers.

You can view the version information of a server (as set in the Project Manager's Build Options dialog) by selecting the server of interest, and then clicking the Version button.

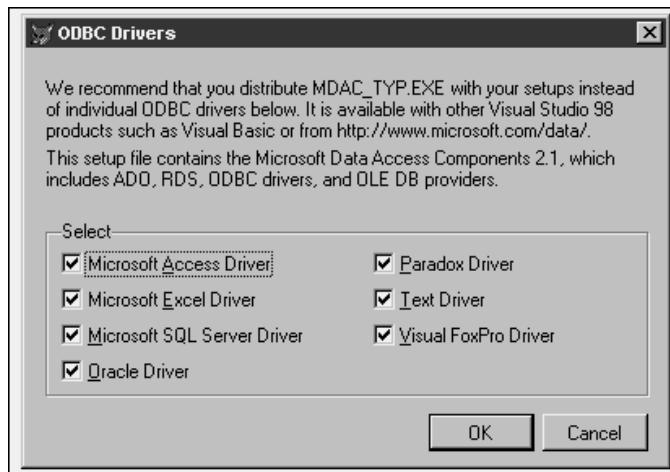
### **ODBC drivers**

If your application talks to a data set through ODBC, you'll want to include the appropriate ODBC drivers. Recently Microsoft started shipping a set of data access components all combined into one setup file, MDAC\_TYP.EXE, and the company recommends that you use it instead of including individually specified drivers.



**Figure 26.7.** Use the COM Component Details dialog to specify options for components to include with your distribution files.

You can find the MDAC\_TYP.EXE setup in HOME() + “\DISTRIB.SRC\SYSTEM”, or in Program Files\Microsoft Visual Studio\VB98\Wizards\PDWizard\Redist, and you can download the most recent version at [www.microsoft.com/data/](http://www.microsoft.com/data/), as shown in **Figure 26.8**.



**Figure 26.8.** The ODBC Drivers dialog is used to identify individual ODBC drivers that you want to include with your distribution files.

If you choose not to include MDAC\_TYP.EXE, you can select individual ODBC drivers you want to include by selecting the ODBC drivers check box in the Step 2 dialog (as shown in Figure 26.4) and then checking the appropriate drivers in the ODBC Drivers dialog, as shown in Figure 26.8.

I can't provide specific suggestions on which way to go because I've only used the individual ODBC drivers up to this point.

### **Microsoft Graph 8.0 runtime**

If your application uses Microsoft Graph, you'll need to include the runtime for it just as you include the VFP runtime for your VFP application. Check this check box if you need it.

### **HTML help engine**

If you include HTML help files (.CHM files) as part of your application, you'll need to include the HTML help engine—which you can think of as a .CHM file runtime. The HTML help engine is comprised of several files, and checking this box is supposed to ensure that all of them get included. I know, I'm making it sound like it doesn't actually do it—I've run into cases where it doesn't, but haven't been able to consistently reproduce the problem. And from the messages posted on the various electronic forums, I'm not alone. These same files are already present on a computer that has Internet Explorer 4.01 or higher. As you know, it's getting increasingly difficult to find a computer that hasn't had IE already installed on it through one means or another. It is an integral part of the operating system, you know.

Nonetheless, some users prefer not to install IE, and so you'll need to include the HTML help engine so that your .CHM files will operate properly.

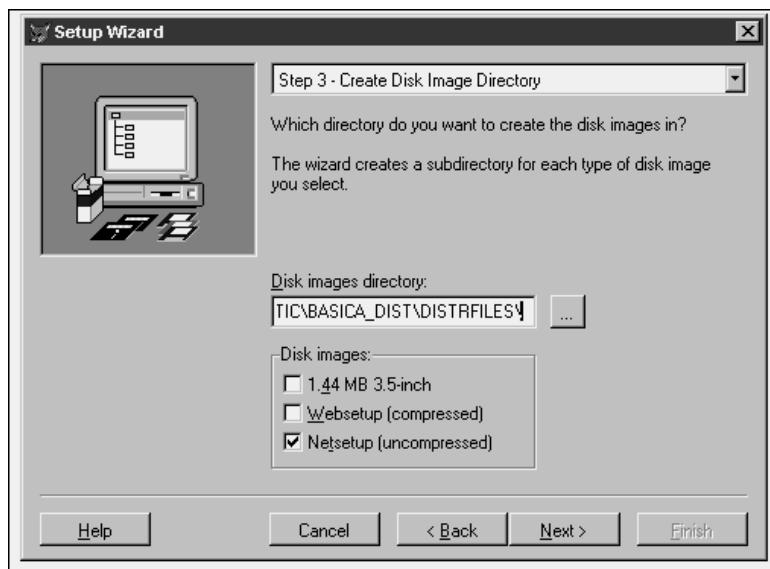
### **Step 3: Create disk image directory**

The third step is to tell the Setup Wizard where the distribution files—the “disk images”—should go. You can manually enter the directory or use the ellipsis button to navigate to the proper directory. In **Figure 26.9**, I used the ellipsis button but then scrolled through the text box so you could see where I was pointing. (The text box isn't big enough for a directory structure of more than about 30 characters.) You kind of wonder if the same programmer wrote steps 1 and 3 in this wizard, eh? You can also choose how the distribution files will be packaged, and I'll cover those in the next section.

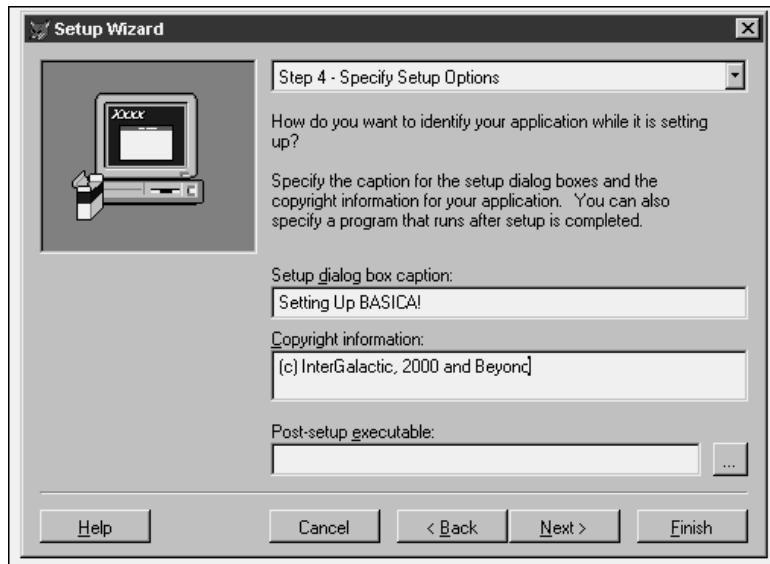
### **Step 4: Specify setup options**

The fourth step is to customize the dialogs that will appear to your users when they run your installation routine, as shown in **Figure 26.10**.

When you're entering information into the caption text box, you'll want to be aware of where that text is used. In Figure 26.10, I used a whimsical phrase that you might think of using yourself. However, the description of “Setup dialog box caption” is a bit misleading, because this value is also used as an adjective and as a noun inside paragraphs in the dialogs that your user will see when running your installation routine, as shown in **Figures 26.11, 26.12**, and **26.13**.



**Figure 26.9.** Identify where to place the new distribution files, and specify how those files will be packaged.



**Figure 26.10.** Specify the Setup dialog box caption and other options.



**Figure 26.11.** The Setup dialog box caption is used as an adjective and as a noun in the Welcome dialog that your user sees when running your installation program.

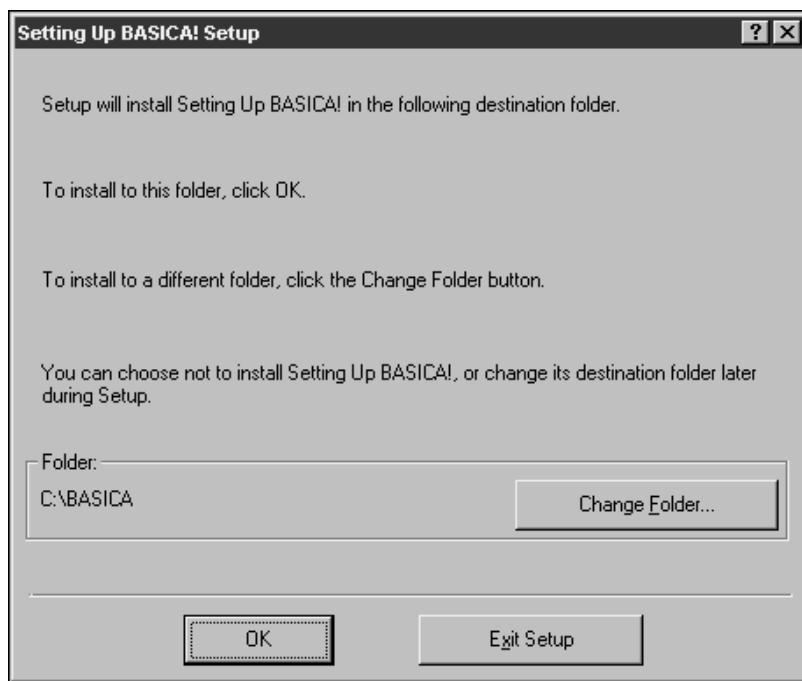
Thus, as repulsive as it may be to you, I would suggest a safe and boring route—that you use just the name of your application—the most creative you can get is to use an exclamation mark at the end of the name.

I would also suggest, however, that you include the version number of your application in the Setup dialog box caption because this phrase is also placed in the “installed software” list box found in the Add/Remove Programs Properties dialog (found under the Start, Settings menu option). In **Figure 26.14**, I show you how the caption I used in Figure 26.10 appears. If it is possible that your users could install more than one version of your application, you’ll want to be sure to identify which version is which here.

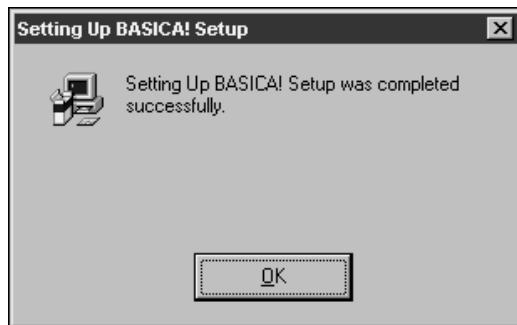
### **Step 5: Specify default destination**

Okay, back to the wizard. It is highly unlikely that your user is going to want to install their program in the same directory structure that you have it in on your development machine. Thus, the Setup Wizard gives you the ability to identify the name of the default directory that the installation routine will attempt to use.

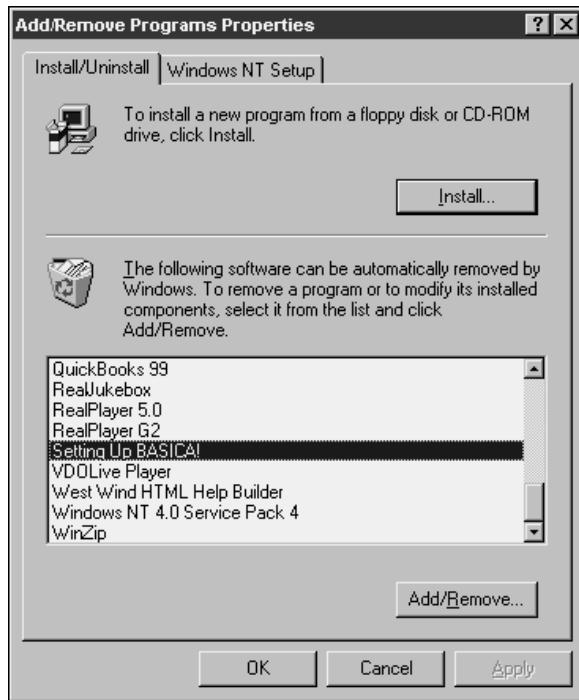
However, when you get to this step, the Setup Wizard will use the lowest level of the directory you specified in Step 1—in this case, BUILDFILES—as the default value for the Default directory text box. In **Figure 26.15**, you’ll see I have already changed the directory to BASICAI. I also created a program group for the company name.



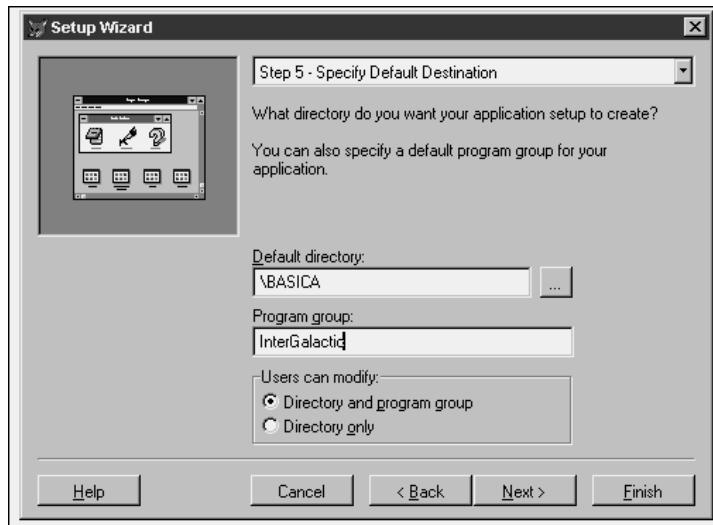
**Figure 26.12.** The Setup dialog box caption is also used as a noun in the Destination dialog that your user sees when running your installation program.



**Figure 26.13.** The Setup dialog box caption is used as a noun in the Completion dialog that your user sees when running your installation program.



**Figure 26.14.** The Setup dialog box caption is also used in the Add/Remove Programs Properties dialog.



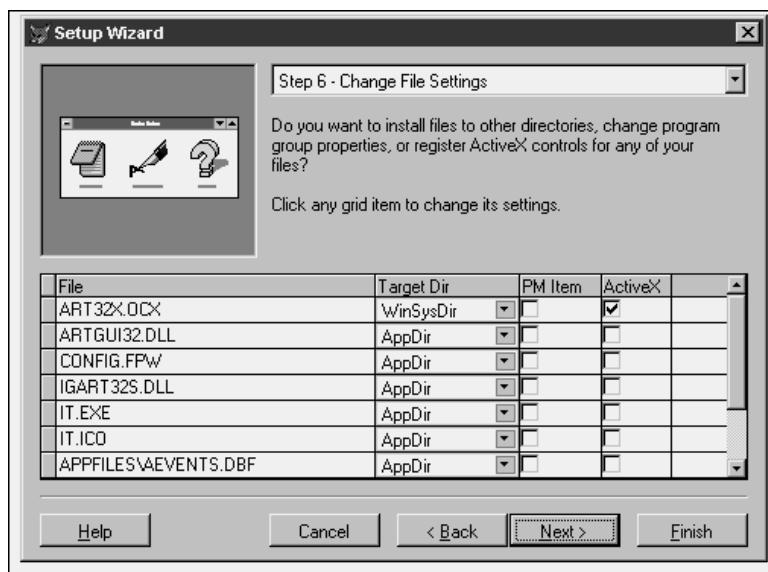
**Figure 26.15.** Specify the destination directory for your application.

You can control whether or not the user can modify one or both values. Figure 26.12 shows the user's view of the values I used in Figure 26.15.

## Step 6: Change file settings

The sixth step is to change any file settings for the files you are going to distribute. The grid in the dialog in **Figure 26.16** contains every file in your distribution directory, and the wizard has made a guess at (1) where to install the file, and (2) whether or not the file is a Program Manager item or an ActiveX control that needs to be registered with Windows during installation.

I've found this dialog makes pretty good guesses, but you might want review the choices if you've got an unusual or complex set of distribution files.



**Figure 26.16.** Here you can choose to install files in other directories and identify files as ActiveX controls.

## Step 7: Finish

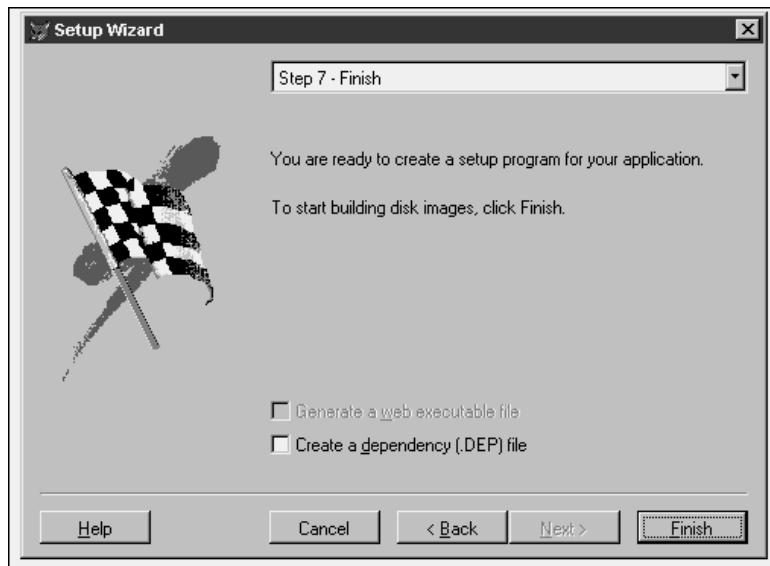
Finally, the last step—in the Setup Wizard, that is—is to click the Finish button, and sit back and wait, as shown in **Figure 26.17**.

You'll get a WAIT WINDOW message:

```
Please be patient while files are being compressed.
```

and a dialog with several items and a thermometer bar scrolling across the dialog for each step. It can take several minutes to the better part of a half hour to build the distribution files, depending on the size of your app and the horsepower of your machine. The process takes

longer the first time because the Setup Wizard has to compress a couple of files that it will then cache for future builds.



**Figure 26.17.** Click the *Finish* button to let the Setup Wizard do its thing.

## Stats of the build

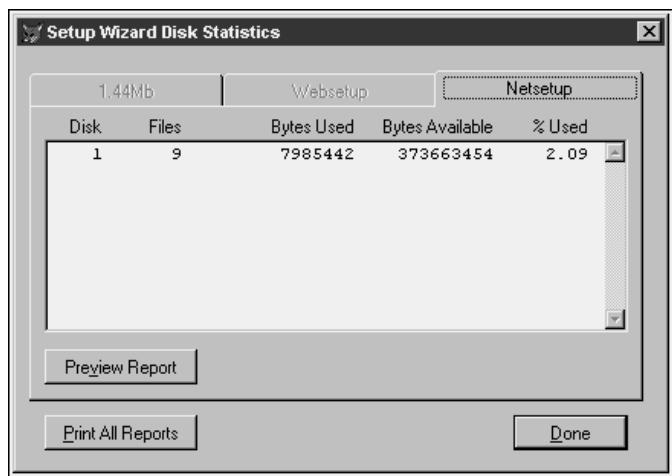
When the Setup Wizard is done, you'll get a dialog like **Figure 26.18**. If you've opted for multiple distribution packages, the appropriate tabs will be enabled and you'll see the stats for each.

You can print a report from this dialog if you want to archive the information or use it as part of the documentation for your application.

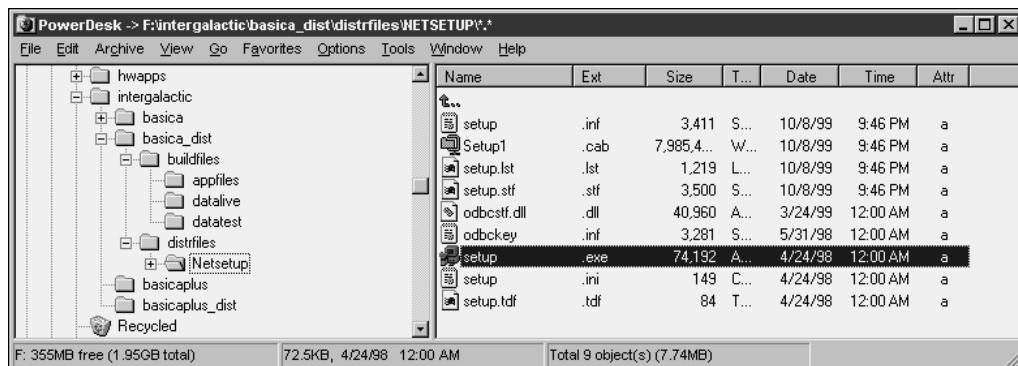
## Where are the results?

You'll have one or more directories under the directory that you specified in Step 3. For example, if you selected the Netsetup option in Step 3, you'll have a directory called NETSETUP under the DISTRFILES directory. Those are the files you'll distribute to your users. In **Figure 26.19**, I've shown the contents of a simple Netsetup distribution directory.

The results vary according to which check boxes you selected in Step 3 of the Setup Wizard. The 1.44 MB 3.5 Disk Images check box will create a series of directories under a main directory, each named DISK01, DISK02, and so on, with each directory full of compressed files up to a maximum of 1.44 MB. You can then copy each of these directories to individual floppies, should you be required to distribute your application this way.



**Figure 26.18.** The Setup Wizard Disk Statistics dialog shows the stats of how many files and how much disk space was consumed.



**Figure 26.19.** The contents of the NETSETUP directory at the end of the Setup Wizard's processing.

Websetup will create a single directory, with all of the files compressed. This saves time when downloading from a Web site, which can be important for users still using a modem or otherwise slow connection, but the flip side is that the installation time is longer, because the files have to be decompressed during installation.

Netsetup will create a single directory with all of the files, but uncompressed. This allows the user to run the installation from this directory and do it as speedily as possible, since the files don't have to be uncompressed. You can also use this mechanism if you're distributing your distribution files on a CD-ROM (and your application fits on a 650 MB CD).

A new Visual Studio installation routine was released during the Fall of 1999, but it's not quite ready for prime time for VFP apps yet—I suggest you wait until Visual Studio 7.0, when it's more tightly integrated with the development tools.

# Section VI

## Visual Studio Topics

It used to be that FoxPro was a standalone tool, used for building database applications that were deployed on a single desktop computer or over a LAN. And you could do anything and everything just with Fox.

But times have changed. Ya gotta have more capabilities than what Fox by itself offers. And if those capabilities are already available elsewhere, why reinvent the wheel and build them into Fox again? Just as Microsoft Office has become a suite of office-related tools that work well with each other, Visual Studio is striving to become a suite of development tools that each does its own thing, and that work well with each other as well. The plan is that each tool in Visual Studio can be used to build parts of an application for which it is particularly well suited.

In this chapter, I'm going to introduce you to three pieces of Visual Studio and the VS development environment that you – as a Fox developer - should start getting to know. First, I'm going to editorialize and discuss my take on where the Windows DNA strategy is today, why I don't think it's ready for prime time yet, and some basic definitions that seem to have gotten lost in all the excitement. Next, I'll introduce you to two key technologies – ADO and MTS – that you need to be aware of. You probably won't use them, but you'll want to watch how they progress, and having a good grounding in the basics will help you understand their progress. Finally, I'll provide a jump start to Visual Basic – as Fox developers, you're often hampered by seeing scads of examples with source code – all in Visual Basic. A little bit of familiarity with VB will help you take advantage of this wealth of information.



# Chapter 27

## Weird Question Time

The current pitch coming from Microsoft's app dev groups is "Windows DNA"—an architecture for creating and deploying distributed applications across multiple computers of various platforms. You can read all about it elsewhere, such as at the Microsoft Web site and just about every piece of marketing literature that comes out of Redmond these days.

But you have to decide whether this is for you. In talking to developers all over the world, I've heard that many of them feel like they're the only developer on the planet who hasn't deployed a half-dozen multi-tier applications using COM+ on the Windows 2000 Beta yet. Well, 90% of you can't all be in the "last 10%." It's important to provide some perspective on all the hype you're exposed to. In this chapter, I'm going to provide you with my perspective—my opinion—on the state of things at the end of 1999. I'm also going to pull together a miscellany of definitions and other items that don't seem to be covered anywhere else, and that don't seem to fit anywhere else.

My physics teacher in high school stood in front of class on the first day and said, "Remember those weird questions you used to ask your parents? The ones like 'If I'm in an elevator that's falling to the ground, and I jump up at the last minute, will I get killed?'" Remember how your parents used to say, "I'm busy" in response? Well, now's the time to ask those weird questions.

And now here's the time to ask those questions you didn't want to ask anyone else, because you thought you were the only one who didn't know the answer.

### Should I or shouldn't I?

The next few pages may get me excommunicated as far as Microsoft (and other vendors involved in the distributed arena, should any of them mistakenly pick up this book) is concerned, but I have to be blunt—I'm not a big fan of this Windows DNA and distributed computing stuff. At least not yet. Here's why.

First, it feels a lot more like marketing than anything with real substance. Sure, you've seen the demos: How to build a mission-critical, enterprise-wide application supporting hundreds or thousands of people—in two and a half hours. Uh huh.

The situation is that LAN applications are old news. We know how to build them, and there isn't a lot of hoopla around them anymore. People take them for granted. Client/Server also had its day in the sun, but that's yesterday's news as well. Product vendors (and the industry rags) need something new to blab about. Three-tier, N-tier, Web-enabled, distributed—these are the hot buttons now.

But I don't see a lot of people doing it yet. Sure, I'm in Milwaukee, a good place to be when the end of the world comes, cuz everything happens here 10 years later than anywhere else. But I talk to people all over the world, and while I see bits and pieces of N-tier development happening, it's not mainstream yet, like, say LAN applications or Web

applications. We're talking about cutting-edge tools, technology, and techniques—in use by very few people so far.

Second, the architecture is not sound. Reliability is haphazard; error handling is abysmal; processes are continuously being redefined. If you're undertaking application development in this environment, you're not engineering solutions, you're performing R&D and building "one-off" models that will need constant care and supervision. My telephone line has been on, with the exception of physical interruptions like lightning strikes or substations that were flooded, for 40 years. My Web server needs to be rebooted every week or two, and my desktop computer, with the sixth set of fixes to the operating system installed, still GPFs every day.

The most common phrase you'll see in Microsoft marketing literature these days is "quickly and easily." Sure, for that two-and-a-half-hour demo. But real-world Windows DNA projects are not quick and they're not easy. Well, using today's crop of visual tools is quicker and easier than coding it all in Assembler. But that's about it. These applications are complex systems that take a long time and are hard to design and build. These are enterprise-wise systems that run entire companies. They're not mailing lists. They need to be engineered properly, with reliable tools and repeatable processes.

I've used this term—engineer—for a very specific reason. An editorial in *Computing Canada* in 1997 made this observation:

"IT projects are not engineering projects. Software research is a better description. In engineering, a repeatable process is applied to a problem. The technology stays the same, the tools the same, the process stays the same. If any of these change, the project goes over budget, or risks failure. With software projects, the technology changes every project, the tools change every project, and the process is in a continuous state of flux. Add to this mix that what is likely being built is the automation of a poorly understood system. All that should be known is that with all the unknowns, any estimates are educated guesses."

I include this statement in every proposal I submit to a customer.

The track record of application-architecture design in our industry has been uneven. Look at DLL Hell, for example. The progenitor of Windows DNA, the .DLL architecture was supposed to reduce or eliminate multiple applications duplicating the same functionality through component reuse. What actually happened has caused countless hours of lost productivity as developers and network administrators try to reconcile issues caused by incompatible .DLLs.

Who in the military-industrial complex of the 1970s would have imagined a worldwide electronic commerce system being built upon the Internet? As a result, a lot of patches and kludges are being used day to day because the foundation was not intended for the current use.

And what about ADO—the current incarnation of data access from Microsoft? It's now in its fifth or sixth major release—in less than three years. How are you ever going to build something reliable when the foundation keeps changing more often than the seasons?

I'm not blaming these folks—who knew back then what we'd be doing now? Ten years ago, no one had ever even heard of a terabyte. Last weekend, a bunch of college students just threw together a 2.1-terabyte system of Linux boxes—just to see if they could do it. But the fact remains that the architecture and tools aren't ready for truly well-engineered solutions.

---

The general approach to developing an architecture has been to throw stuff against the wall and see what sticks, using incessant revisions to fix the big things. This is a strategy introduced by Netscape, where you could live for years on one beta release after another, never having to actually buy a real product. But it bodes poorly for having a stable platform upon which you can develop systems for your customers and users.

Third, there are no comprehensive design tools that can be used to engineer a complete solution. We're being asked to build mission-critical solutions that span the enterprise, and yet we still have a hodgepodge of design and analysis tools that don't work together. Indeed, the toolbox for analysis and design processes that most developers have access to contains Word 97 and a demo copy of Visio.

In addition, the majority of developers building applications are self-taught, poorly trained, and have pulled themselves up by their bootstraps. There are no industry-wide credentialing programs or requirements as there are for medicine, law, accounting, or architecture. Thus, even if there were clearly defined processes and tools, the users—developers like you and me—wouldn't be required to use them, or be monitored as to whether we were using them correctly.

Finally, Microsoft's investment in software has slowed drastically. Most of their investments over the past couple years have been in Internet-related areas, not in the software development arena. The strategy was at one point to help developers create Windows-based applications that spanned the enterprise, so as to expand the reach of Windows throughout the enterprise as has already happened with the desktop. But this may be an afterthought anymore. Application development is clearly not a high priority. Defects continue to be a major issue. With the resources at Microsoft's disposal, they could have a revolutionary effect on defects—both with their own tools, and as a market leader that could influence the rest of the industry, but as I just explained, they won't. As a result, we are left with tools and architectures that are deemed "good enough." Not reliable, not repeatable.

While none of these is bad in and of itself, together they tell me that the WinDNA strategy isn't ready for repeatedly building reliable applications upon.

Nonetheless, though I don't think this stuff is ready for prime time yet, there are others who disagree. As a result, a couple of my authors are looking at a couple of Windows DNA-based books for mid- to late 2000, for example. And there are instances where there are significant business reasons to deploy now, even though the technology isn't stable. They're willing to put in the extra time needed to fine tune and baby sit these types of apps, because, for their particular business or application, it makes economic and competitive sense. So let's open the door, and see what's around the corner.

## Where does VFP fit?

I hope you don't have to read this section, but I've included it just in case. Where does Visual FoxPro play in the bigger scheme of things?

First of all, it's still the best tool for developing high-performance desktop and LAN database applications. Period. The native data engine, the visual tools, and the rich programming language and object model—all these combine to make VFP the Dream Team for app dev on the desktop.

It's also a stupendous tool for two-tier applications—it has the same rich tools for developing a front end, and native hooks to attach to back ends as any other tool out there.

As far as three-tier, the position that VFP is supposed to commandeer is that of the middle tier, building business objects that talk to a variety of front ends, such as browser interfaces and UI's built in other languages; process operations and mash data; and then move data back and forth to a data store, such as SQL Server. Again, VFP's rich language features and speed make it ideal for performing these types of tasks.

There's a white paper on the Microsoft Web site—Visual FoxPro Strategy Backgrounder—that details this extremely well. Grab a copy and keep it with you when meeting with customers! To get there, point your browser to [msdn.microsoft.com](http://msdn.microsoft.com). Then on the left side, select Products, Visual FoxPro, Product Information, Future Directions. You should also be able to search on “Strategy Backgrounder” if the site has been reorganized between the time I wrote this and the time you read this.

## What is a type library?

You've probably seen the term “type library” in your various readings, either in an online help file or in other documentation—most often in conjunction with something to do with Visual Basic. And you've probably shrugged it off—you've been able to get through hundreds of pages of developing database applications without hearing about it once. But you're going to hear more and more about type libraries, and eventually you're going to run into a situation where you're going to be asked, during the creation of a component, or the registration of an object, for the type library—or be told that the type library can't be found.

A type library is a binary file with an extension of .TLB that contains the definition of the interface of a COM component. This sounds like I snarfed this out of a C++ manual, so let me explain how you'd use it.

First, though, I want to reiterate what I mean by the term “interface.” Many of us are still used to thinking of an interface as that picture on the monitor. That's the user interface. In this context, I'm talking about the programmatic interface—the collection of properties and methods that are visible to other objects.

When you create a COM component, your goal is for other folks to use it. And for those other folks to use it, they're going to have to know the properties and methods—well, the public properties and methods. How do they find out? If they're lucky, you've included documentation and examples, like so (I'm using VFP syntax because you know it):

```
o = createobject("mycomponent.myclass")
m.lcName = o.GetName()
```

The user knows that there is a method called GetName because they read the documentation. But how does the actual software know? The type library acts as the software's documentation for the interface—the list of the object's properties and methods that are available to the outside world.

Thus, when another object wants to access your component, it's got a method call, like o.GetName, in its program code. However, just because that string of text is in its program code doesn't mean that the object, o, actually has a method called GetName(). The type library tells the calling object that the component that GetName() is okay.

An analogy is if you went into an apartment building looking for Apartment 4102 because someone told you to. You don't know if 4102 is actually a valid apartment number, though, do

---

you? You have to go to the apartment and look at the directory to see if 4102 is listed. The directory is the same as the type library—it tells you what is actually valid.

When you create a VFP COM component, a type library is automatically created. In VFP SP3 on Windows NT, the type library is also automatically included inside the .DLL or .EXE file, which means you don't have to distribute a second file. In earlier versions of VFP or on Windows 95/98, you'll need to distribute the .TLB and .VBR files as well.

Custom user-defined properties and methods are included in VFP type libraries (remember, the type library is simply a “listing” of properties and methods that are available to the outside world) as long as they are defined as PUBLIC. Visual FoxPro type libraries also include a return value type and a list of parameters (all are defined as variants), and property descriptions (if you were good developer and entered descriptions for your properties!).

How does this work inside? Don't know, don't care. But that's why COM components—even COM components created with VFP—need type libraries. Most of the time, as with Service Pack 3 of Visual FoxPro 6.0, the type library details are taken care of for you.

## What is a GUID?

In the Build Options dialog (see **Figure 27.1**), you'll notice a check box that says “Regenerate Component IDs.” Hmm. What does that do?

Your next step is to open the help, only to find this less-than-lucid explanation:

“Installs and registers Automation servers contained in the project. When marked, this option specifies that new GUIDs (Globally Unique Identifiers) are generated when you build a program. Only those classes marked OLE Public in the Class Info dialog box of the Class menu will be created and registered. This option is enabled when you have marked Build OLE DLL or Build Executable and have already built a program containing the OLEPublic keyword.”

Aha! It makes sure new GUIDs are created! But what's a GUID? For the longest time, I thought it meant Gooey, Uooey, Icky Database. Who could have guessed it had to do with computers? Well, maybe you did. So you did a search on “GUID” in MSDN online help (I told you to install the whole thing, didn't I?), and the top-ranked topic is...

### **SQL OLEDB Provider Events for Visual Studio Analyzer**

Helpful, eh? Well, scroll on down a bit more, and find this definition:

“An identifier used to precisely identify objects and interfaces. All Windows applications and OLE objects have a GUID that is stored in the Windows registry.”

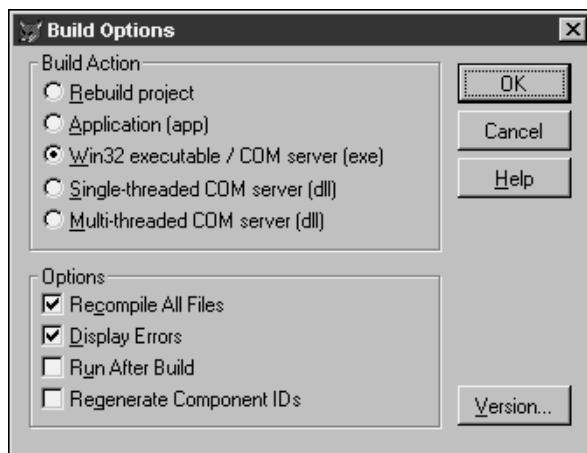
So it appears that when you build a project with an OLE Public class (one that will become a server that needs to be registered with Windows), you may want to create a new ID in the Windows registry. But nowhere in the doc does it say *why* you might want to—or not want to! Here's the deal.

First of all, this check box is enabled only when you want to build a server. You want to check this box when you change the programmatic interface to your server. In other words, if you add or delete public properties or methods, you'll want to have a new GUID generated.

Why? Well, this server is going to be called by another component or object—and you don't know what that is going to be. When you change the interface, that other component or object doesn't automatically know about the change. In fact, when that other component is built, the type library for your server is included in the component. This is how that other component knows what the interface to your server is—and what the GUID for the server is. If you change your server's interface but not the GUID, how is that other component going to know that the interface changed? It won't—it will still use the old type library, which points to the same old GUID, but also describes an interface that is no longer valid. What's the most likely result? GPF city, dude!

When you change the GUID for your server, the type library bound into the other component will now point to a GUID that is no longer valid, and you'll get a nice friendly message like "Server not found." Instead of a GPF.

It actually gets a bit more interesting—see the next bit on early versus late binding.



*Figure 27.1. The Build Options dialog with the potentially mysterious Regenerate Component IDs check box.*

## What's the difference between early and late binding?

If you're hearing or reading about GUIDs and type libraries, you're probably being exposed to terms "early binding" and "late binding" as well. This is clearly another topic that Visual FoxPro developers have been able to live without for a long time—it really smells like a "C" kind of thing—best left alone in many people's minds. But as you get into the world of components and servers and ActiveX controls, you're going to be hearing about this stuff.

---

When you create an executable, a lot of things are included in the file. As Fox developers, we have been accustomed to ignoring all that—we just waited until the thermometer bar finished scrolling across the screen, and then ran the Setup Wizard.

Different tools perform the executable-building process in different ways.

There's a bunch happening behind the scenes, however, particularly if you include COM components and ActiveX controls. For example, the type libraries for these components can be built right into the executable. In other words, suppose your component—your client—makes method calls to “another server.” If, while you're building the component, you can determine what object the methods belong to, the reference is resolved at compile time, and your component includes the code necessary to invoke the server's method.

This makes performance pretty fast—the executable can get to the memory address for a specific method call immediately because it's part of the executable. The call overhead can be a significant part of the total time required to set or retrieve a property, and, for short methods, you can also see huge benefits. This is called early binding. Some people use the term “vtable binding.”

However, if you don't build the type library of the component into the executable, you have a more flexible architecture—your executable can read the address of the method at run time. It's slower, but you can make the address dynamic. (Actually, you wouldn't yourself, but the tools you use would do so.) This process—not including the type library into the executable, and reading the addresses at run time—is called late binding. Some people call this “IDispatch binding.” The executable is “binding” to the addresses as late as possible—when the app is actually executing.

Thus, particularly with early binding, you'll want to change the GUID for a server when you change the interface, because the actual addresses that are bound into the executable are very likely going to be wrong—so your executable is going to make a call to a routine in memory that isn't there. Yes, that's pretty close to a guaranteed GPF.

Put another way, binding (in general) describes how one object can access properties and methods of another—specifically, how a client can access the interface elements of a server. Early binding provides better performance, but at a loss of flexibility.

## What is Visual InterDev?

Visual Studio consists of about a half-dozen tools: Visual Basic, Visual C++, Visual FoxPro, Visual InterDev, and Visual J++. You know what VB and C do, and you probably don't care what J++ does, although you probably guessed it has something to do with Java or the Web. But what's this InterDev? Sounds ... cool. Well, sorta confusing, too. Maybe you even opened it up and goofed around a bit. And the question on the tip of your tongue is still, “What is it?”

Well, we can figure this out. “Inter” must have something to do with the Internet! And “Dev” must be for developers, right? Internet Development? OK, we guessed that much already. But what does it do? Does it replace Front Page? Maybe it's a second Java language development environment? Or ... what?

It's a development environment—a very, very cool development environment—for putting together database applications that are deployed on the Web. Now, I know you're thinking that there are a lot of tools that claim to do this, so it might be hard to differentiate.

Let me draw an analogy. In the olden days, in order to create an application, you used a file editor—or, more accurately a text editor, to enter a series of commands into a text file. Then you would use a development tool to link libraries with the text file:

```
C:>emlink myprog.txt lib1.lib, lib2.lib, lib3.lib
```

Then you would compile that text file, usually through a command line:

```
C:>emmake myprog.txt myprog.exe
```

The libraries would automatically be included. Then you would run that executable, sometimes with an additional runtime, and sometimes not:

```
C:>MYPROG
```

Then development tools became more sophisticated. They bundled a text editor and some other tools in an integrated package, and you could do the text editing, the linking, the compiling, and even run the executable all within that same program. And these development environments—or “IDE” for Integrated Development Environment—became more and more sophisticated. You could open multiple text files, create lists of subroutines, open link files, store libraries, and so on.

With Windows and other GUI environments came a whole new set of tools that allowed you to create visual components as well as text files, and perform all sorts of related tasks, all within the same IDE. Whether you’re using Visual C++, Visual Basic, or Visual FoxPro, life as a developer is pretty good, compared to the old days.

The development of database applications that you want to deploy on the Internet, however, sent developers back to the days of multiple programs, each good for just one thing. One tool was used for creating HTML, another managed site diagrams, a third provided scripts, another deployed files from the development machine to the live Web server, and yet another dealt with the actual databases. And, of course, there were a lot of discontinuities between the tools. There had to be a better way—fortunately, it didn’t take 20 years to get to that better way.

Visual InterDev is an integrated environment that does all of this, plus lots, lots more. You can create Web pages, work with style sheets, create components, write scripts, open and manipulate databases, create site diagrams six ways from Sunday, deploy your applications, and even shell out to other programs. And about another million functions that I haven’t mentioned.

It’s probably the single coolest program with the most potential in Microsoft’s stable. It’s just awesome. Unfortunately, like much of what Microsoft sends out the door, it’s not quite ready for prime time. As of version 6.0, it’s very slow, requires huge amounts of resources, and the various pieces are still buggy and don’t work well with each other. However, if you recall, the same could be said of many other early Microsoft programs. If they continue to put reasonable resources towards improving VID, it will be the killer tool for the early 2000s.

## What's a ProgID?

Sounds like a user-friendly name for a GUID, doesn't it? Well, kinda sorta, but not really. When you create a component and register it with Windows, several entries are made for the component. One is the GUID—that 16-byte value that uniquely identifies the component. However, if you're writing a program and want to refer to that component, you probably don't want to use the GUID. Instead, you can refer to a second entry in the Windows Registry (called a ProgID) for the component.

A ProgID is the “user-friendly” name of the component that you use to instantiate it. The ProgID takes the form of myobj.myclass. That's all!

## What does reentrant mean?

You've probably heard the mantra, “Builders are reentrant, wizards aren't.” But that doesn't tell you much, other than possibly hinting at the idea that builders are cooler than wizards.

Reentrant, a term borrowed from C++, actually means something slightly different in VFP. In C++, when a program was reentrant, it meant that a second user could run it, and the second user wouldn't interface with the first user. In other words, user A is running a routine, and is currently executing line 20. Then a second user runs the same routine, and is on line 14. If the program is reentrant, there is no conflict. This sounds like a definition of multi-user, but in this case, it's really referring to a blob of code.

In VFP, being reentrant simply means remembering the state when it's run a second time.

## What's the difference between “in-process servers” and “out-of-process servers”?

An in-process server—a file with a .DLL extension—is run in the same memory space, and requires fewer resources. It's also faster, because there's no communication needed between one process and another. But crashing the app brings down the server, and crashing the server will likely hose the app. And an in-process server can't be deployed remotely.

Out-of-process—with an .EXE extension—is run in a separate memory space and requires more resources. But it can be deployed remotely. And crashing the server doesn't crash the app, or vice versa. Not every .EXE is an out-of-process server, though. You have to include an OLE Public class in your executable in order to expose functionality of that class to other applications. The very cool thing is that you don't lose any of the normal functionality of your application—the rest of your users may not even know that other applications (not just other users) are calling the application as well.

## Where do I go next?

The single best resource to learn more about these things is the *Visual Basic Programmer's Journal*. While it focuses on VB, it has a ton of useful articles on topics relating to Windows, components, servers, and other Visual Studio-oriented topics. And you might find some of the VB articles interesting as well; the most common use of VB (unbelievably) is to build database applications. You'll learn things about user interfaces, and chortle uncontrollably at the basic information presented about data, but all in all, it's a great read.

You might also try *MIND: Microsoft Internet Developer* (although they may have changed the name again since this writing). This publication has lots of good stuff—although much of it is slanted for the C++ crowd. There's usually at least one excellent, on-target article each month.

There is a little-known help file for Service Pack 3 of Visual Studio. Look for VFPSP3.CHM in the root of your Visual FoxPro directory (most likely Program Files\Microsoft Visual Studio\VFP98). This file contains a bunch of good information—when you're finished with this chapter, check it out. You'll probably learn much more from this additional help file.

# Chapter 28

## An Introduction to ADO

**ADO, short for ActiveX Data Objects, and an associated technology, OLE DB, are the current incarnations of Microsoft's data access technology. Visual FoxPro developers, having a blindingly fast native data engine and easy access to back ends like SQL Server and Oracle, have long turned a blind eye to this side of Microsoft's offerings. In a multi-tier world, however, you might need to expand your horizons a bit. In this chapter, I'm going to give you a quick tutorial on ADO.**

VFP's data access is pretty darn great—it's fast, easy to use, and pretty stable. The basic concepts—record pointers navigating through a set of relational tables—haven't changed for 20 years. But not all data in the world out there is stored in .DBFs; in fact, there's an awful lot of data that isn't stored in relational tables at all.

Microsoft has been working on the problem of providing access to this entire universe of data—and doing so in a consistent manner. The technologies they've been working on through the 1990s are starting to mature, and the current version, Universal Data Access, is implemented through a pair of acronyms: ADO and OLE DB.

### A (very brief and very rough) history of the Microsoft data access strategy

I'm going to skip over the burning questions of the moment—What is ADO and why do I care?—to give you a bit of perspective on the data access strategy coming out of Redmond.

In years gone by, Microsoft was well known for not doing data well. Until the early 1990s, in fact, after nearly 15 years in business, they still didn't have a database product of any sort. They had an excellent word processor, a great spreadsheet, a pretty good presentation tool, and so on. But the one big hole in their suite of desktop applications was any sort of database tool.

Rumor has it that they had made several attempts in the late '80s and early '90s, and each, for one reason or another, never matured enough to see the light of day. Finally, a new group put together a product called Access, and this tool was intended to do two things. First, it was going to provide a mechanism to access and manipulate a variety of data from all over the organization—thus, its name. Second, it was going to provide a mechanism for desktop PC users to work with databases of their own. Access was going to be a new part of the office suite of tools, next to Word and Excel. And for this purpose, Access needed its own database engine. Thus, JET was born.

As has been repeated ad nauseam since its creation, JET was not named because it was fast. Rather, JET is an acronym for "Joint Engineering Technology." Yes, it was a database engine designed by a committee, and the early versions definitely demonstrated all the attributes accorded to something designed by a committee. Nonetheless, as the Microsoft Way dictates (you didn't think that One Microsoft Way was a street address, did you?), the team persevered and solved many of the problems of the initial engine: speed, multi-user access, flexibility, and so on.

While this was happening, another branch of Microsoft went out and bought Fox Software, acquiring a popular and powerful database tool for the application-development side of the business. Also coming along for the ride, or perhaps the underlying reason for the acquisition, was a top-notch development team and a rabid development community.

Microsoft was finally getting serious about data.

The next step, however, was not to simply build on these two platforms, but to figure out a broader strategy for gaining access to (and, eventually, controlling) all of the data at the desktop, and then the company. A new mechanism was needed.

The database engine development teams iterated several times, and if you've done any Visual Basic work or know any Visual Basic developers, you've probably heard at least part of the litany of acronyms. The most recent, before ADO and OLE DB, was DAO (Data Access Objects), a series of OLE controls that VB developers used to connect their applications to databases. But this wasn't enough—it wasn't fast or flexible enough.

Another, grander mechanism was needed. Enter OLE DB, stage left.

## **OLE DB (and ADO)**

In order to explain where OLE DB fits in, let me take you back to the days of printer drivers. Back before Windows, it was a lot more difficult to talk to printers from various applications. Initially, you had to put special codes, called control strings, in your application to make a printer do anything more than print Courier 10 point. And each application required you to do this differently. It was really hard, it was boring work to figure out how to do it, and it didn't work very well. And the worst part was that if you needed to send output to a different type of printer, you had to go through the whole process again, using the special codes for *that* printer.

Then someone came up with the idea of creating "drivers" for printers. Instead of programming special codes, your application would come with a set of intermediary programs, each of which knew how to translate output from your application into the codes that a specific printer required. Thus, your application would have one driver for a wide-body Epson, another driver for that brand new Hewlett-Packard LaserJet, and a third driver for the fancy Sony label printer. Your app talked to the driver, and the driver talked to the printer. What could be easier?

Windows, of course, then put a common (and easy-to-use) interface on this process, but the idea remained the same.

Talking to data, on the other hand, is still sort of back in the days of inserting special codes in each program. Your application generally knows how to talk to only one type of data. Visual FoxPro reads .DBFs. Excel reads .XLS files. SQL Server opens .MDF files. And none of them reads anyone else's data—at least not very well.

Imagine if you had a set of drivers that acted as intermediaries between applications and data. And all you had to do was tell your application to use a certain driver and the app could then access data from a variety of sources. Wouldn't that be neat? Well, we've already got it in place, in a limited fashion. This is what ODBC does, for a few well-defined types of data. You can create a data source with an ODBC driver, and then configure your application to use that data source. Your application can then access the data just as if it belonged to the app itself.

There are a number of technical issues and process-oriented issues that limit ODBC in manners such that it wasn't the be-all and end-all of data access. But it was a good start, and is in widespread use today.

What a developer would really like, instead of just a dumb connection to selected data, is a more robust and flexible application that managed multiple “drivers” that could talk to a wide variety of data. Not just relational data sitting in Oracle, SQL Server, and Excel, but also data in e-mail systems, binary data like multimedia files, and random textual data like in desktop organizers.

And this driver manager application would be able to manipulate that data with a common lingua franca, so that whether your program was Visual FoxPro, Visual Basic, Excel, or a script on a Web site, the data could be accessed the same way.

This driver manager application is essentially what OLE DB is—and the drivers are called “providers.” The only trouble is that using OLE DB is kinda hard. It’s a sophisticated, low-level interface, and requires a lot of work to communicate with. What we really need, then, is a user-friendly interface to OLE DB.

And that’s what ADO is—an easy-to-use interface for OLE DB.

## What is ADO?

So what is ADO, anyway? I don’t mean, “What does it do?” And I don’t mean, “Where is its place in history, or in the data access strategy hierarchy at Microsoft?” I mean, what—physically—is it? When you hear the answer, you’ll probably respond like I did when I realized that classes were just .DBFs, or like Charlton Heston did when he discovered that Soylent Green was just people. No big mystery—ADO is just a .DLL. Well, actually it’s a bunch of .DLLs, but that’s splitting hairs.

Since it’s just a .DLL, you instantiate ADO just like you instantiate any other COM component. Once you’ve done so, you’ve got access to its properties and methods. However, in the case of ADO, those properties and methods allow you to identify and attach to a data source, select from that data source a subset of data (the subset is called a recordset), and then manipulate—navigate and maintain (add/edit/delete) the recordset. You can almost think of ADO as being a component that provides an objectified Visual FoxPro cursor—a cursor with properties and methods as well as fields and rows. In fact, the folks who wrote ADO nearly all have Visual FoxPro data engine experience somewhere on their resume. Pretty darn neat, eh?

So it’s no big deal, and that’s why this basic fact seems to be overlooked a lot. Every discussion of ADO just jumps into working with it. Kind of like the computer instructor who launches into what a computer is, and how to use the software on it, without ever showing you where the power switch is.

## Where do I find ADO on my computer?

The ADO .DLLs are part of the Microsoft Data Access Components (MDAC) and get installed on your machine for all sorts of reasons, kinda like Internet Explorer. MDAC comes with the new flavors of Windows, with Visual Studio, and with developer versions of Office. And, in the odd happenstance that you still don’t have MDAC, you can download the latest version (and bug fixes) from the Microsoft Universal Data Access Web site at [www.microsoft.com/data](http://www.microsoft.com/data).

Not sure what you have? Do a file search in Windows Explorer for “ADO” and “MDAC” on your C drive. You should find a few ADO .CHM (compiled help) files, and an MDAC directory or two.

Depending on what you've got installed on your machine, you might have more than one version of ADO. As of this writing, the current version of ADO is 2.1, Service Pack 2.

## **What are the previous versions of ADO?**

Version 1.0 of ADO was released in early 1997, and was designed primarily for use with Active Server Pages. This was quickly followed by 1.5, which included both new functionality and additional providers for ADO.

Version 2.0 of ADO was worthy of an increment in major version number, because it provided the ability to create client-side recordsets, filters and indexes, and recordset sorting. Starting to sound like VFP cursors, eh?

Version 2.1, the current version, includes the ability to save client-side recordsets as XML documents.

Because these new releases have been introduced rapidly one after another, you very well might find MDAC 1.5 and MDAC 2.0 files on your machine along with 2.1 versions.

## **Getting started with ADO**

I know, I know. You want to see some code. Well, we can do that! First, let me explain what you need to follow along, and the steps you're going to go through.

If you're in front of your computer, you'll need Visual FoxPro 6.0, SP3 (duh!), the latest version of MDAC, and some data. You can use either Visual FoxPro's TasTrade sample database or the Pubs database in SQL Server 7.0. You can also use another data source, but you'll have to substitute the appropriate names and IDs for the ones I use.

## **Bare-bones steps for accessing data through ADO**

Once you've got your machine set up, here are the steps you'll go through:

1. Create a Data Source Name on your computer.
2. Instantiate ADO.
3. Connect to the data.
4. Create a recordset.
5. Manipulate the recordset.
6. Wrap a user interface around ADO.
7. Go home.

Step 6 is there for a reason, by the way. You might end up having so much fun that you forget to go home. I don't want to get in trouble with your family by keeping you here late.

## Create a Data Source Name

This is one of those topics that may seem a little awkward to pure VFP developers. A Data Source Name (DSN) is a file on your computer that identifies how to connect to a data source somewhere on your network. For example, suppose you're on your workstation and you want to connect to a SQL Server database on another machine. You'll need to know the following pieces of information: the name of the machine that the SQL Server is running on, the name of the SQL Server database, the username, and the password.

Once you've connected to the database, you'll need to know more stuff, such as the structure of the database files—the names of tables and fields, and perhaps something about the data inside the database.

A DSN helps you with the first half of this. It contains all this information in a single, easy-to-use place.

You can create three types of DSNs: User, System, and File. Basically, a User DSN is a data source that is available only to a specific user, and a File DSN is a data source for a specific file. While I'm sure there's a good reason for each of them, I've never run into a situation where I needed to use one. A System DSN is a data source definition for the computer, and security across multiple users is handled either by the operating system or the back-end database. So I'm just going to concentrate on System DSNs.

To create a DSN for either TasTrade or SQL Server (the screen shots that follow will use SQL Server), open the Control Panel from the Start, Settings menu, and select the ODBC Data Sources applet. You'll be presented with the ODBC Data Source Administrator dialog shown in **Figure 28.1**.

I'm going to ignore the first tab, User DSN, as well as the File DSN tab, shown in **Figure 28.2**. The action for the rest of today will be centered on the System DSN tab, shown in **Figure 28.3**.

The System DSN tab shows all of the currently installed System Data Sources. In Figure 28.3, you'll see four different data sources: a test DSN for the Pubs database that comes with SQL Server, a test DSN for another SQL Server database named MIDAS, a production version DSN for a new version of the MIDAS database, and a DSN for the Small Business Financial Manager that relies on Access instead of SQL Server. How it got there, I don't know—I don't remember installing something like this. (Have you had experiences like this, too?)

To add another DSN, click Add to open the Create a New Data Source to SQL Server dialog as shown in **Figure 28.4**.

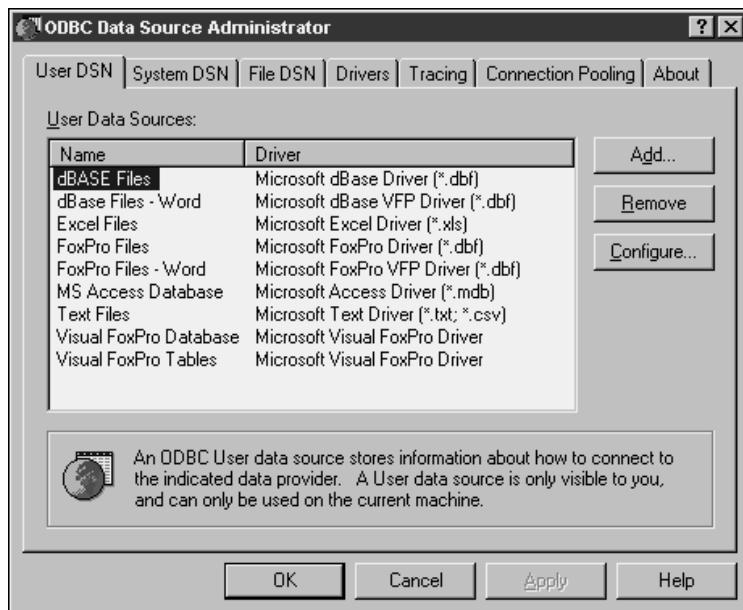
The first few times you navigate through this dialog can be tricky. The Name you use to refer to the data source is the actual text string you'll use as a parameter in your code, like so:

```
oC.open("PubsTest","sa","")
```

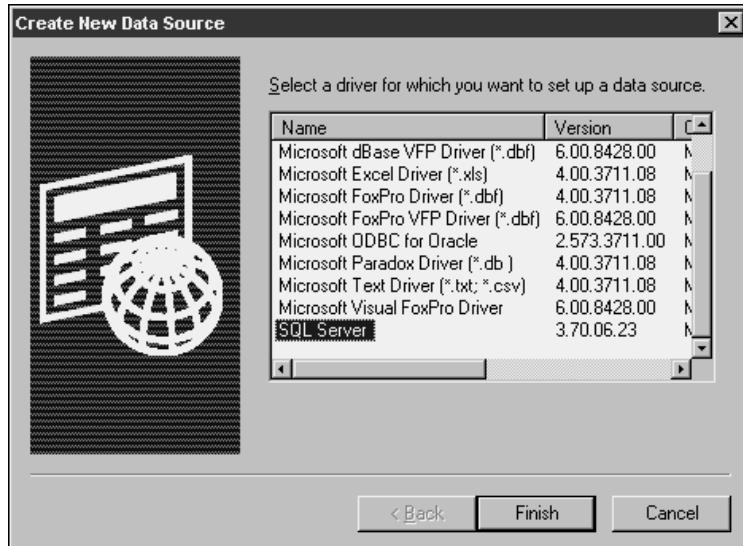
So if you use a moniker like "Herman's Way Cool Sales Database" as the "Name", you'll end up writing code like this:

```
oC.open("Herman's Way Cool Sales Database","sa","")
```

Probably not a good idea. Instead, make the Name short and sweet. You can enter "Herman's Way Cool Sales Database" in the Description text box.



**Figure 28.1.** The ODBC Data Source Administrator allows you to create User, System, and File DSNs.



**Figure 28.2.** The dialog that appears after clicking Add in the File DSN tab of the ODBC Data Source Administrator.

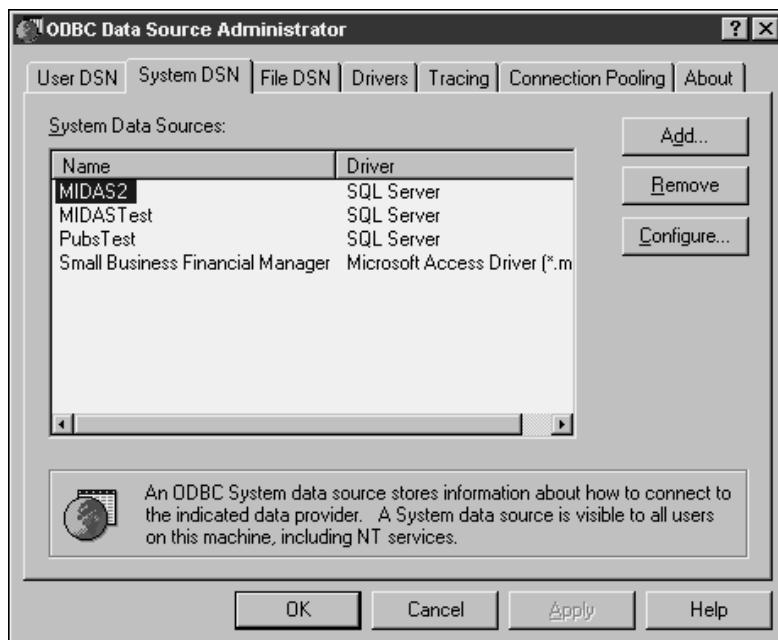


Figure 28.3. The System DSN tab of the ODBC Data Source Administrator.

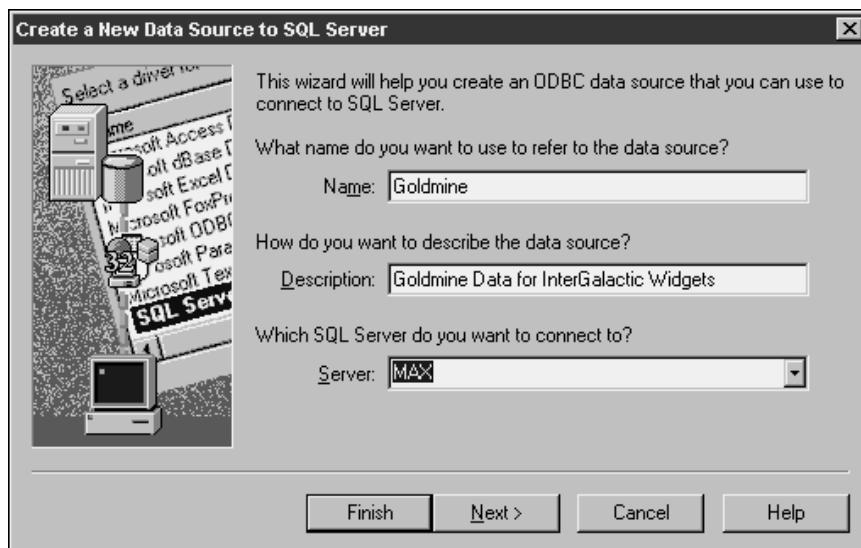


Figure 28.4. Clicking Add in the System DSN tab of the ODBC Data Source Administrator dialog opens the Create a New Data Source dialog.

Finally, pop open the Server combo box to select which SQL Server you want to connect to. If you're new to SQL Server, this can be a bit misleading.

You know how you can use the term "server" to mean either a physical box, like a "file server," or a piece of software, like "Web server"? In this instance, the term "SQL Server" means the program running on a box. In Figure 28.4, I've got a box named "MAX" upon which SQL Server 7.0 is running. If I had another box named, say, ZEUS, with SQL Server running on it as well, there would be two items in the Server combo box, and I'd have to choose which of the two SQL Servers I wanted to connect to. So, to be clearer, the prompt should really say something like "Which instance of SQL Server do you want to connect to?"



*What if you open this combo box and the name of the box upon which your SQL Server is running doesn't show up? Well, you're hosed. The documentation says, "Contact your System Administrator", but, let's face it, you're probably the system administrator, aren't you? When this has happened to me (it really isn't a frequent occurrence, which is probably why there's no help or error handling for this situation), I've ended up uninstalling the SQL Server from the box it was running on, and reloading it from scratch. What else ya gonna do?*

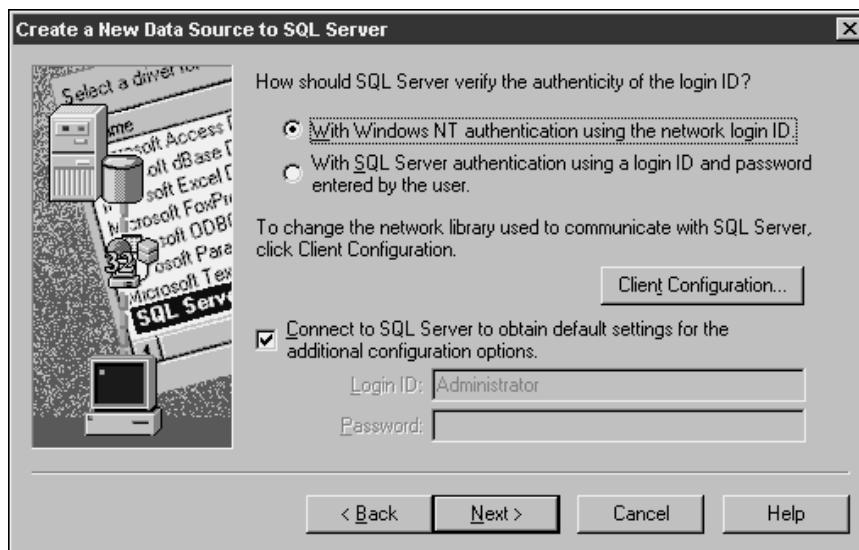


*How do you name your machines? I've found a whimsical naming scheme is difficult to keep track of, so I've decided upon a temporal scheme, where the name has to do with the period of time that the machine was acquired. For instance, the box I used during the VFP 3.0 beta is named TAZ, the box I used during the VFP 5.0 beta is named ROADRUNNER, the 6.0 beta box is named TAHOE, the 7.0 box is named SEDONA, and so on.*

*This method is helpful because it's an instant reminder of approximately how old the machine is. TAZ, for example, has been relegated to the basement, serving as a print server—what else are you going to do with a P90 with 32 MB of RAM? It's also helpful because it means I get to buy a new machine pretty often.*

*Unfortunately, MAX showed up on our doorstep one night when there wasn't a beta in sight, so there wasn't a handy acronym. However, we happened to be watching The Grinch Who Stole Christmas, and thus the machine was named MAX (after the dog).*

Assuming you've found your SQL Server, click the Next button to continue with the wizard. The next dialog, shown in **Figure 28.5**, allows you to specify how to connect to the SQL Server. Whether you want NT authentication to do the work for you, or if you want SQL Server authentication, is up to you and your network configuration and requirements. Go ahead and click Next again.



**Figure 28.5.** The next step in the wizard is to identify how to connect to the SQL Server.

Within a SQL Server installation on a computer, you can have one or more databases. For example, as shown in **Figure 28.6**, on the SQL Server on MAX, I've currently got a number of databases, including model, msdb, Northwind, Pubs, tempdb, and a few more that aren't visible in the default database combo box. Select the database that you want to connect to. In this case, Pubs would be a good choice, because the ADO examples in this chapter all use it.

Again, what if your database doesn't show up in this list? Time to call your SQL Server Administrator—hoping that it's not you. A SQL Server database is just a big file with an .MDF extension, but just copying it to the box you've got SQL Server running on isn't enough. You need to go into SQL Server's Query Analyzer and attach the .MDF within SQL Server. You'd use a command like:

```
EXEC sp_attach_db@dbname=N'YOURDATA', @filename1='e:\yourdir\yourdata.mdf'
```

where YOURDATA is the name of the .MDF file, and so on. Well, this isn't a SQL Server tutorial, but if you get hung up on the initial attachments, a lot of the rest of the chapter isn't going to do you as much good, is it?

Back to the creation of your DSN.

After clicking Next, you'll get to change a bunch of settings in the dialog shown in **Figure 28.7**. I usually blow right by this one.

Clicking the Finish button in the dialog in Figure 28.7 allows you to verify what you've done through a configuration dialog as shown in **Figure 28.8**. I always click the Test Data Source button because I'm still skeptical about whether or not this is really going to work. Kind of like getting ready to go on vacation, getting in the car, driving down the street, and then driving back to make sure, one last time, that you locked the front door.

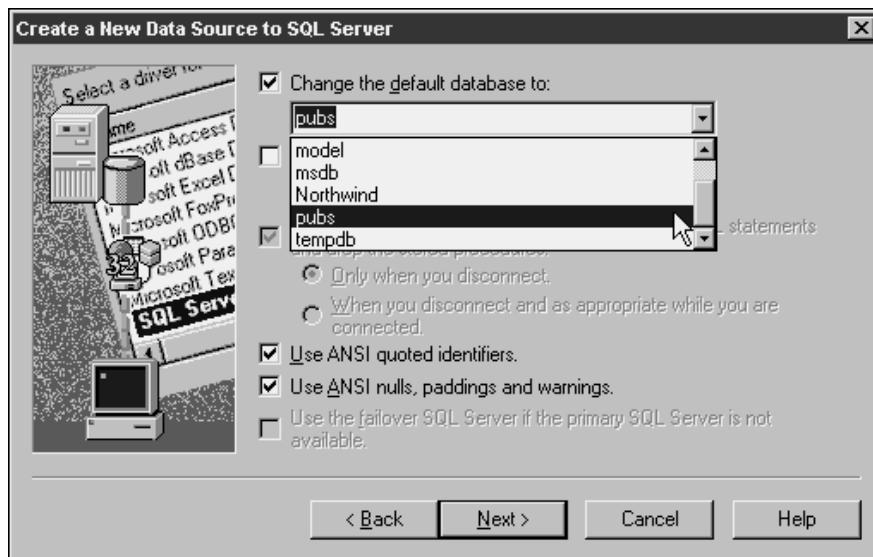


Figure 28.6. Select the default database in SQL Server.

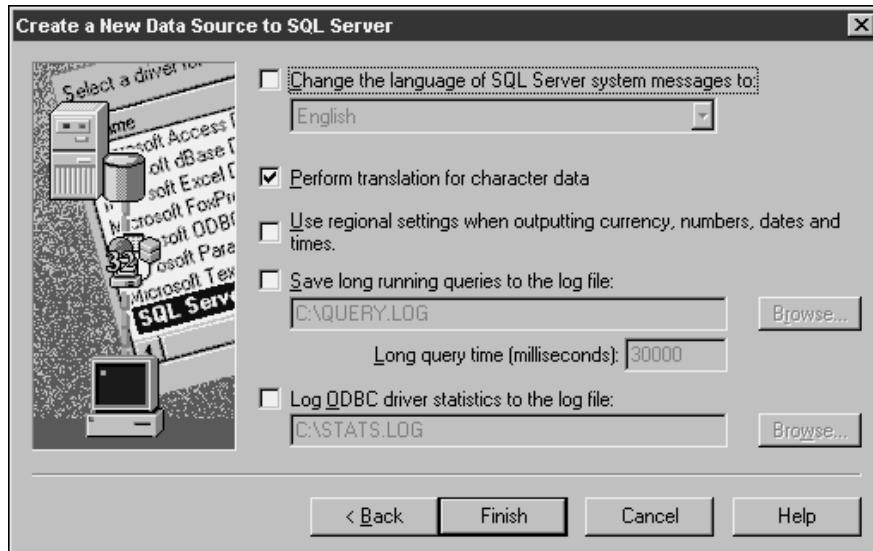


Figure 28.7. More settings to mess with along the way of creating a DSN.



**Figure 28.8.** The ODBC Microsoft SQL Server Setup dialog allows you to verify the settings you've selected and to test the connection.

If the connection works, you'll get a dialog like **Figure 28.9**. I don't know what it looks like if it fails because I don't think the test has ever failed on me.

Click OK in the Test dialog (Figure 28.9), and then OK in the Setup dialog (Figure 28.8), and you'll see your new System DSN in the System DSN tab of the ODBC Data Source Administrator dialog, as shown in **Figure 28.10**.

Now it's time to attach to the data. As I said, I'm going to use the PubsTest System DSN in the following examples. If you've created a DSN to the SQL Server database Pubs and called it "HERMAN," you'll need to make the appropriate substitutions. If you're attaching to a different database, you'll need to also change the code that connects to specific tables and talks to specific fields in the tables. That's what you get for not following along with the teacher.

### Instantiate ADO

 Now that you've got a DSN, you can start to write code. Create a new directory to work in, and create a new program file, say, TEST1.PRG. If you just want to follow along, you can find the source code for this chapter in CH28 in the source code downloads for this book.

If the ADO .DLLs have been properly installed and registered, there's a ProgID (see Chapter 27 for more on ProgIDs and related topics) named ADODB.Connection in the Windows Registry. You can create an object reference to a database connection object with this ProgID. The code that instantiates an ADO object is as follows:

```
* test1.prg
```

```
* creates a connection object
oConn = createobject("adodb.connection")
```

If the ADO .DLLs have not been properly installed and registered, or if you mistyped the command, you'll get an error. One possible error is shown in **Figure 28.11**.

By itself, a Connection object isn't very interesting. Well, maybe a few people on the far end of the pocket-protector spectrum are getting all hot and bothered, but the rest of us would like a bit more.

### Connect to the data

Now that you've got a Connection object, you'd like to use it to connect to a specific data source. Use the "open" method of the Connection object, like so:

```
oConn.open("PubsTest", "sa", "")
```

This passes the PubsTest DSN, along with the default username of "sa" and default password of a blank. If you'd named your DSN "Herman's Way Cool Sales Database" as the DSN Name, you'd use that string instead of "PubsTest."

### Create a recordset

Next, it's time to use this connection to get hold of data. First, create a Recordset object:

```
oRS = createobject("adodb.recordset")
```



**Figure 28.9.** The ODBC Data Source Test dialog.

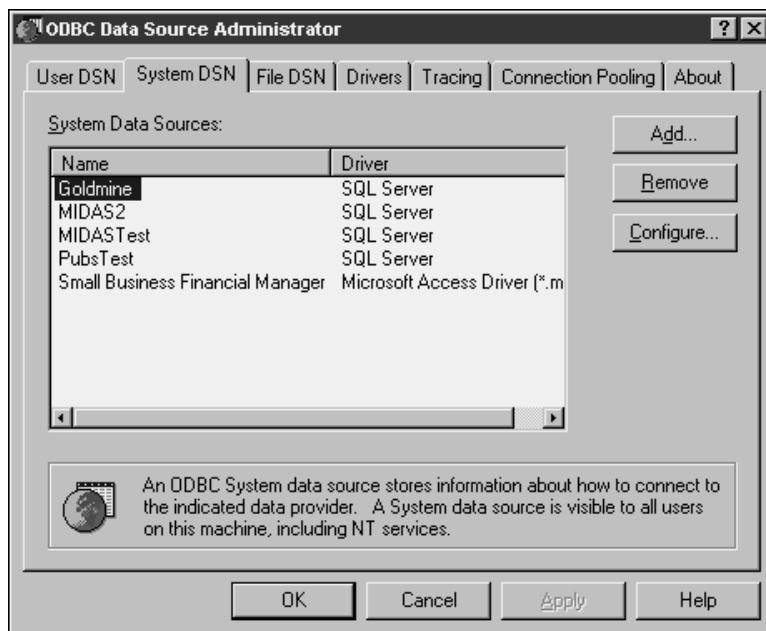


Figure 28.10. The System DSN tab with the new DSN.

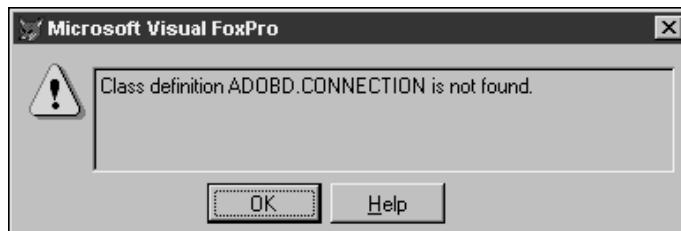


Figure 28.11. Mistyping "ADODB" as "ADOBD" results in an error.

Just like creating the Connection object, the Recordset object is empty until you do something with it. So the next step is to populate the Recordset object, like so:

```
ORS.open("select * from AUTHORS", oConn)
```

The first parameter is a simple SQL SELECT command—note that you need to have some inside knowledge of what the structure is, like the name of the table you want data from. And you pass the object reference of the connection you want to use—you could have several connections open, and so you'd need to specify which connection you want to use. Altogether, the program looks like this now:

```
* test1.prg
* creates a connection object
oConn = createobject("adodb.connection")
oConn.open("PubsTest", "sa", "")
oRS = createobject("adodb.recordset")
oRS.open("select * from AUTHORS", oConn)
```

In Visual FoxPro, the equivalent of these four commands would be:

```
use PUBSTEST
```

### Manipulate the recordset

Now the good part starts. Time to get into the data itself. The first shot at this will simply scroll through the recordset and print the name of the author to the desktop. Here's how:

```
do while !oRS.eof()
? oRS.fields("au_fname").value + " " + oRS.fields("au_lname").value
oRS.MoveNext()
enddo
```

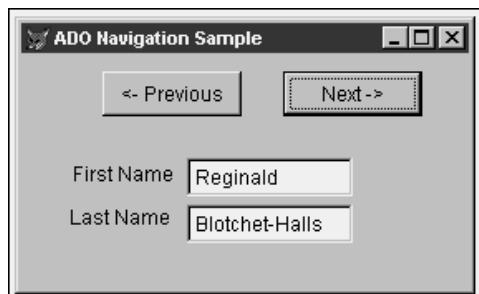
Pretty easy, eh? As a VFP developer, you already know most of this. The EOF() property of the Recordset object makes sense, and the MoveNext() method is also obvious. The syntax of the Fields collection—using the name of the field inside parentheses, and then identifying which property (in this case, the Value property)—is a little strange, but easy enough to understand.

And that's all there is to it!

### Wrap a user interface around ADO

You're probably going to have to search high and low to find a user who would put up with a database program with no user interface, so how about wrapping a form around this code? Once the form has a handle on this-here recordset, you can display the information to the user in a series of text-box controls, and give users the ability to navigate through the recordset themselves. Golly, the more things change, the more they stay the same, don't they?

The screen in **Figure 28.12** shows a simple navigation form that displays the first and last name for an author in the PUBS AUTHOR table.



**Figure 28.12.** A sample ADO Navigation form.



To create this form (it's called TEST2.SCX in this chapter's source code), create an empty form, and add two command buttons (cmdNext and cmdPrevious) and two text boxes (txtFirst and txtLast). Add a method called MapControls, and a property called oRSAuthor. Then stuff the following code in their respective methods:

```
* init()
oC = createobject("adodb.connection")
oC.open("PubsTest","sa","");
thisform.oRSAuthor = createobject("adodb.recordset")
thisform.oRSAuthor.open("select * from authors", oC, 1)
thisform.mapcontrols()

* mapcontrols()
thisform.txtFirst.value = thisform.oRSAuthor.Fields("au_fname").value
thisform.txtLast.value = thisform.oRSAuthor.Fields("au_lname").value

* cmdNext.click()
if !thisform.oRSAuthor.eof()
  thisform.oRSAuthor.movenext()
endif
thisform.mapcontrols()

* cmdPrevious.click()
if !thisform.oRSAuthor.bof()
  thisform.oRSAuthor.moveprevious()
endif
thisform.mapcontrols()
```

Now it's time to explain what's going on.

The code in the Init() looks mostly familiar. The first difference is that the object reference in the recordset creation is assigned to a form property, not simply to a memory variable as it was in TEST1.PRG. If it had been stored to a memvar, the memvar would have disappeared as soon as the Init() was finished, right? By assigning it to a form property, the entire form has access to the record set.

The MapControls method, first called from the end of the Init(), simply binds data from specific fields in the recordset to the text boxes. Sort of a hard-coded ADO version of SCATTER MEMVAR. You'll notice that you can reference the members of the Fields collection of the recordset regardless if the recordset reference is stored as a memory variable or a property. Yeah, takes a bit of getting used to, doesn't it?

The Next and Previous command buttons' Click() methods are no big deal. First, the position of the recordset "record pointer" is checked to make sure that you're not at beginning or end of file, using the BOF and EOF properties of the recordset. Hmmm, BOF? EOF? Sound familiar? Has Wayne Ratliff (the creator of dBASE II) infiltrated the ADO development team? Or have the good ideas from Visual FoxPro folk taken hold on the ADO team? If the record pointer was actually moved, the MapControls method is called to update the values of the text boxes.

And that's all she wrote. Well, actually, that's not all. There's one little difference in this code when compared to the TEST1.PRG code, and it's in the call to the recordset Open()

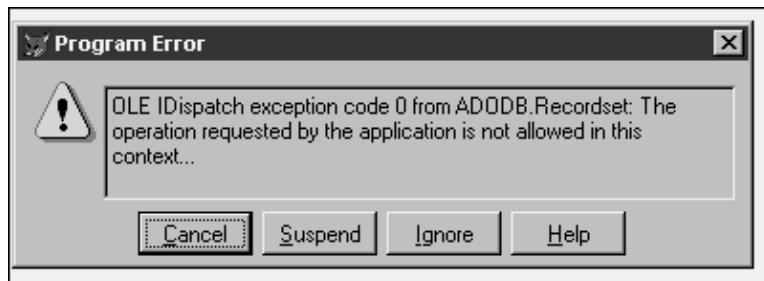
method. Notice the third parameter: “1.” Obviously it’s important, or I wouldn’t have put it there, or begun to make such a big deal out of it here.

Unlike VFP, where a cursor is a cursor, there are four types of ADO recordsets. If you don’t specify which type you want to create, through the use of the third parameter in the Open() method, a “forward-only” recordset is created by default. And, as you can surmise, you can’t move the record pointer backward through the recordset in a forward-only recordset. The “1” parameter creates a Keyset record set, which allows navigation in both directions.

If you’d like to see what happens if you don’t specify a non-forward-only recordset, drop the “1” parameter from the Open() method and run the form again. You’ll be able to click Next until you reach the end of the recordset, but the first time you click Previous, you’ll get a friendly warning as shown in **Figure 28.13**.

The MovePrevious method is not allowed with a forward-only recordset, and generates an error message.

Now, before I wrap up, I’d like to mention one more thing. You’ve probably jotted down a number of questions in the margins as you’ve been reading through this. How did I know to use a “1” for a Keyset recordset, for example? And what are the other types of recordsets? How did I know any of this syntax, in the first place? Good questions, all of them. I’ll cover the answers to these, and other interesting ADO tidbits, in the next section.



**Figure 28.13.** Trying to call the MovePrevious method in a forward-only recordset makes for an unhappy application.

## Go home

Umm, do I really need to include more details in this step?

## The ADO object model

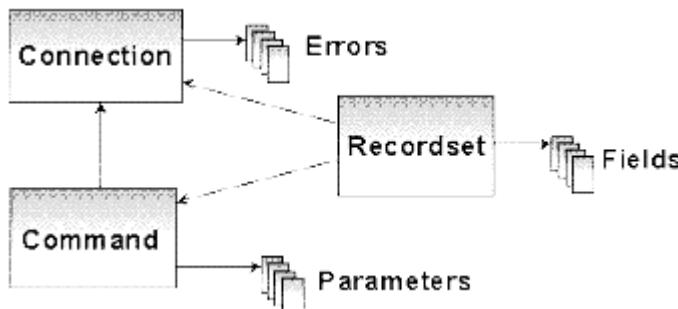
In the olden days, a piece of software would come with a manual and a quick reference card, and you could use the card to find out what language elements you had available to you, and the reference manual to find out more details about using those elements. ADO just shows up on your drive—where do you get the scoop on the language elements?

First, just about everything that comes from Microsoft anymore has an online help file in the form of Compiled HTML Help, so if you passed by its reference in the Readme file, just do a search for ADO\*.CHM on your drive.

There are also a number of books on ADO that cover the mechanism in a fair amount of detail. They usually use Visual Basic code in their examples, but the ADO syntax is pretty easy to noodle without being a VB guru.

But even with these resources, you'd probably like a short introduction to the object model for ADO, and in terms that a VFP developer can get into quickly. I'll cover the biggies here.

The diagram in **Figure 28.14** shows the primary objects and collections in the ADO object model.



**Figure 28.14.** The primary objects and collections in the ADO object model.

Access from your application to a data source is made through a connection. You can think of a connection as a “pipe” through which data is transferred. In fact, one of those troughs attached to a cement truck might be the best analogy, because not only do you have to have the trough, but you also have to point the trough to where you want the cement (or data) to go.

A command issued across a connection manipulates the data source in some way—either navigating through the data or maintaining (adding/deleting/editing) it. Commands aren't islands unto themselves; often you have to include variable data with the command. This variable data is a *parameter*. For example, you might want to apply a filter against a data source. The application of the filter would be a command, while the specifications of the filter (all names beginning with “A”, all balances over \$100) would be contained in parameters.

Commands that return data place that data in local storage called a recordset, much like a SELECT statement in Visual FoxPro creates a cursor. As I mentioned before, you can think of a recordset as being an objectified VFP cursor—with properties and methods.

A recordset consists of one or more rows, and it's important to note that there isn't a single object that represents a single row in a recordset. However, a row in a recordset consists of one or more fields, just like any other relational table.

The last object in the whole smash is the Errors collection. Errors can occur for any number of reasons—most often due to the inability to establish a connection, execute a command, or perform an operation (like I showed you when trying to move backward in a forward-only recordset).

## **The Connection object**

The Connection object hosts a single session with a data source. The actual functionality available to the object depends on which OLE DB provider is being used. In addition to configuring the connection, the Connection object can execute a command, obtain schema information about the database, and examine errors returned from the data source with the Errors collection.

## **The Recordset object**

A Recordset object represents the subset of records returned by an executed command, much like a Visual FoxPro cursor. The Recordset object has a mechanism similar to the record pointer in Xbase languages, such that at any time, a reference to the Recordset object is referring to a single record in the recordset.

There are four types of recordsets available, depending on the capabilities of the OLE DB provider. These are:

- Dynamic cursor. This recordset allows you to view changes (additions/deletions/edits) that other users have made.
- Keyset cursor. This recordset acts just like a dynamic cursor except that it only allows you to view edits—not additions or deletions—that other users have made.
- Static cursor. This recordset acts the most like a VFP cursor in that once you've gotten it, the records are yours until you commit changes—and thus, changes that other users make are not visible to you until you refresh the recordset.
- Forward-only cursor. This recordset is just like a static cursor except you can only move forward through the recordset. Sounds silly, right? This can be useful when you only need to make a single pass through a recordset, such as in a report, or when populating a List Box or Combo Box control, because it's faster than other types of recordsets.

Note that (1) not all providers support all types of recordsets, and (2) if you don't specify a cursor type, a forward-only cursor is created by default.

Recordsets support all sorts of great methods and properties. For example, MoveFirst, MoveLast, MoveNext, and MovePrevious do exactly what they sound like. Move is similar to VFP's SKIP command, while the AbsolutePosition and Filter properties also allow you to move through the recordset as you desire.

Update commits changes immediately, while UpdateBatch commits a group of changes upon call. The Status property allows you to determine if any conflicts were encountered while writing changes.

## **The Fields collection**

A Recordset object has a Fields collection that is made up of Field objects. The Field object has a number of properties, such as Name and Value, that return information about the field. As you saw in the example above, the syntax for referencing a specific Field object is a little tricky at first:

```
oRS.fields("au_fname").value
```

But after working with it for a bit, the syntax makes a lot of sense. Other handy properties of the Field object include Type, Precision, and NumericScale, which all return characteristics of the field; DefinedSize and ActualSize, which return the field size and size of the data within the field, and manipulate the contents of fields with large amounts of binary or character data with the AppendChunk and GetChunk methods.

## The Command object

The Command object is a definition of a specific command to be executed against a data source. For example, you can define the executable text of a command, such as a SQL SELECT statement, much like when you assign a SQL SELECT statement to a memory variable, define parameters, execute the command, and control how the command will execute.

## The Parameters collection

The Parameters collection defines an argument that will be used with a Command object. For example, a SQL SELECT statement could use a parameter to define the column name in the fields list and another in its WHERE clause.

## The Errors collection

The Errors collection is the repository for information about problems encountered during a single operation of data access. Each Error object represents a specific provider error, not an ADO error.

## ADO errors

Note that ADO errors are handled by the run-time error handler, not by the Errors collection. For example, an ADO-specific error will trigger a VFP On Error event. However, the error numbers that VFP gets aren't the same as those listed in the ADO online help.

Cool enough, though, you can use AERROR to trap these errors and deal with them in your error handler. For example, you could set up an error handler like so:

```
* test3.prg
* generates and traps an ADO error
on error do handler
* click on Next a few times, then click on
* Previous to generate an error because TEST3.SCX
* has a Forward-only cursor
do form TEST3
return

PROCEDURE Handler
*
* quick and dirty error handler hijacked
* from the Hacker's Guide to Visual FoxPro 6.0
* (c) Tamar Granor and Ted Roche
*
* Handle errors that occur
```

```
* In this case, we'll just figure out which kind they are
* then save the information to a file

LOCAL aErrData[1]

= AERROR(aErrData)
DO CASE
CASE aErrData[1,1] = 1526
  * ODBC Error
  WAIT WINDOW "ODBC Error Occurred" NOWAIT
CASE BETWEEN(aErrData[1,1], 1426, 1429)
  * OLE Error
  WAIT WINDOW "OLE Error Occurred" NOWAIT
OTHERWISE
  * FoxPro error
  WAIT WINDOW "FoxPro Error Occurred" NOWAIT
ENDCASE

LIST MEMORY TO FILE ErrInfo.TXT ADDITIVE NOCONSOLE

RETURN
```

## Dealing with constants

Remember back when I used a parameter of “1” to open a recordset as a Keyset recordset so that navigation could occur in both directions? Well, if you look in the ADO help file (ADO210.CHM in WINNT/HELP) for the recordset Open method (under the ADO API Reference, ADO Objects, Recordset Object topic), you’ll see the syntax for the command as well as several tables that describe the allowable parameters. For example, the syntax is:

```
recordset.Open Source, ActiveConnection, CursorType, LockType, Options
```

The third item in the list (after Source and ActiveConnection), CursorType, is defined as an optional parameter that determines the type of cursor that the provider should use when opening the record set. It can be one of the following constants:

| Constant          | Description                                 |
|-------------------|---------------------------------------------|
| adOpenForwardOnly | (Default) Opens a forward-only-type cursor. |
| adOpenKeyset      | Opens a keyset-type cursor.                 |
| adOpenDynamic     | Opens a dynamic-type cursor.                |
| adOpenStatic      | Opens a static-type cursor.                 |

The problem is that you can’t plug “adOpenForwardOnly” into a Visual FoxPro statement and expect the statement to execute properly. As you know, a constant is a mnemonic text string that stands in place of a numeric (or other data type) value that might not be as easily recognizable. So while the syntax:

```
thisform.oRSAuthor.open("select * from authors", oC, adOpenKeyset)
```

is a lot easier to understand than:

```
thisform.oRSAuthor.open("select * from authors", oC, 1)
```

it won't work in VFP. Of course, you could define a VFP constant, either in your code or in a VFP header file that is included in the appropriate place in your project, like so:

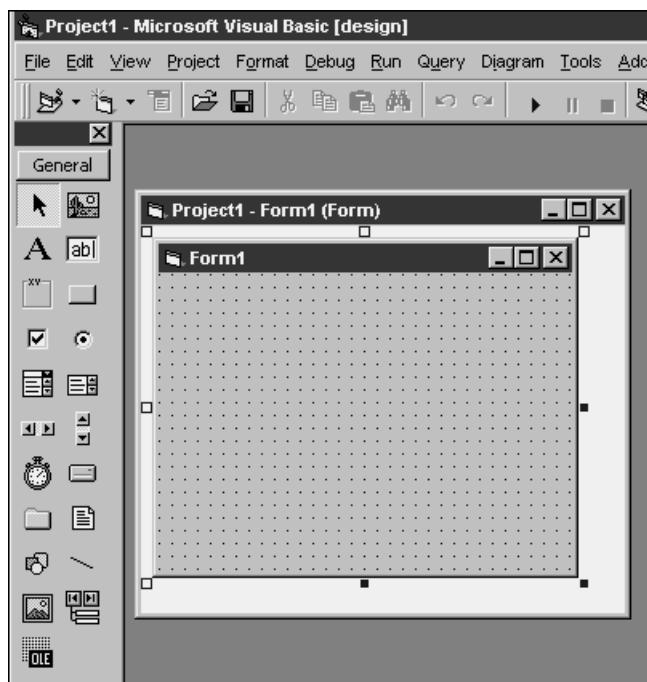
```
#DEFINE adOpenKeyset 1
```

However, the problem remains: How do you know that adOpenKeyset is equal to 1? The ADO documentation just uses references to constants, with nary a reference to actual values.

As with many things in life, there's an easy way and a hard way to find out the enumerated values of these constants. The easy way is to find the ADOVFP.H include file, which is in the Program Files\Microsoft Visual Studio\VFP98\Tools\Xsource\VFPSource\Wizards\Wztable directory. (You might have to unzip the XSOURCE ZIP file in the VFP98\Tools directory.)

You can also roll up your sleeves and do a bit of spelunking via the Visual Basic Object Browser. While not necessary, being comfortable with the Object Browser will come in handy one day, and the better acquainted with it you are, the better off you will be.

Open the Object Browser in Visual Basic and inspect the ADO object, examining the global definitions for the constants you're interested in. To get a handle on the ADO object, open Visual Basic, create a new "Standard EXE" project, and select a form, as shown in **Figure 28.15**. You should also have a toolbox labeled General with a selection tool and 20 controls.



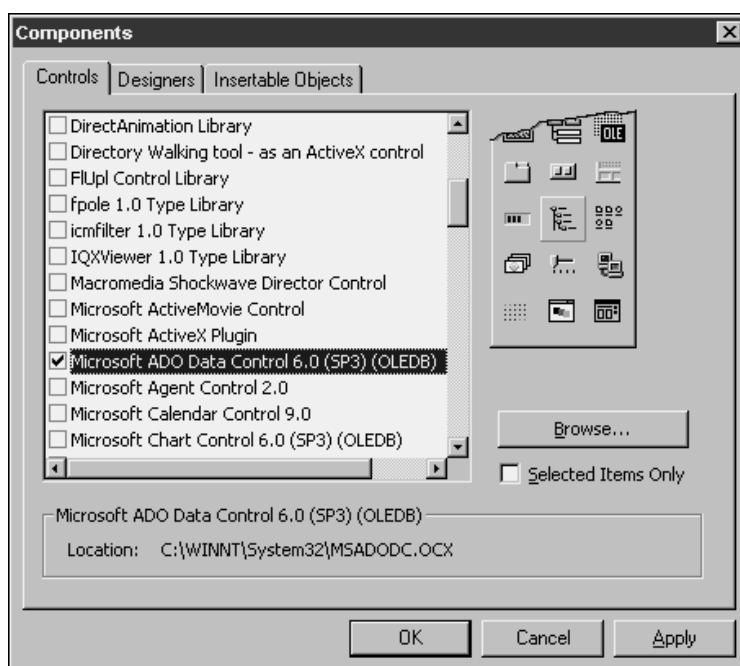
**Figure 28.15.** An empty Visual Basic form with the standard set of controls in the General toolbox.

Right-click in an empty area in the General toolbar and select the Components menu option to open the Components dialog. Scroll down the list until you get to the Microsoft ADO Data Control 6.0 (SP3) (OLEDB), and check the check box on the left, as shown in **Figure 28.16**.

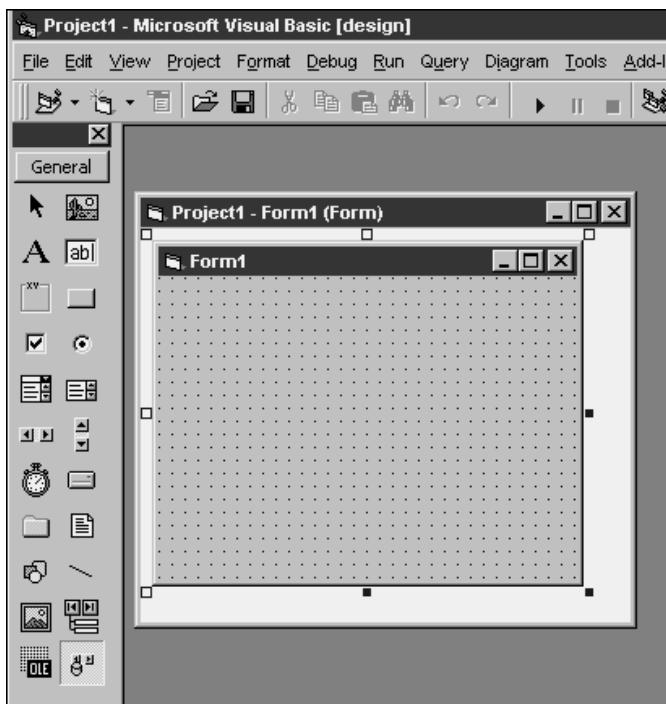
Close the dialog by clicking the Apply and OK command buttons, and you'll see the new control added to the General tab of the toolbox, in the lower right corner, as shown in **Figure 28.17**.

Now pop the control onto the form by selecting the control in the toolbox, and then clicking and dragging the control's outline on the form as shown in **Figure 28.18**. Note that simply clicking on the form isn't enough—you have click and drag the control to place it on the form. Alternately, you could just double-click the ADO control in the toolbox, but the control will be placed dead-center in the form, which, in another lifetime, might not be where you want it.

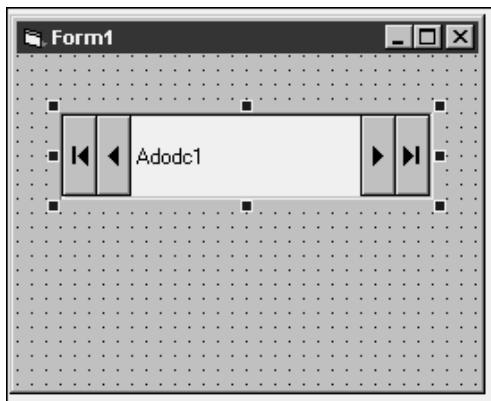
Yeah, this is a lot of work just to find out what a constant is, isn't it? Well, it is the first few times, but once you see the nifty Object Browser that's part of Visual Basic, you'll pine for one in VFP. Here it is. Select the control on the form (as opposed to the form itself), and then either press F2 or select the View, Object Browser menu command in Visual Basic. The Object Browser will open. Pop open the combo box of libraries as shown in **Figure 28.19**, and select the ADODB library.



**Figure 28.16.** Selecting the Microsoft ADO Data Control in the Visual Basic Components dialog.



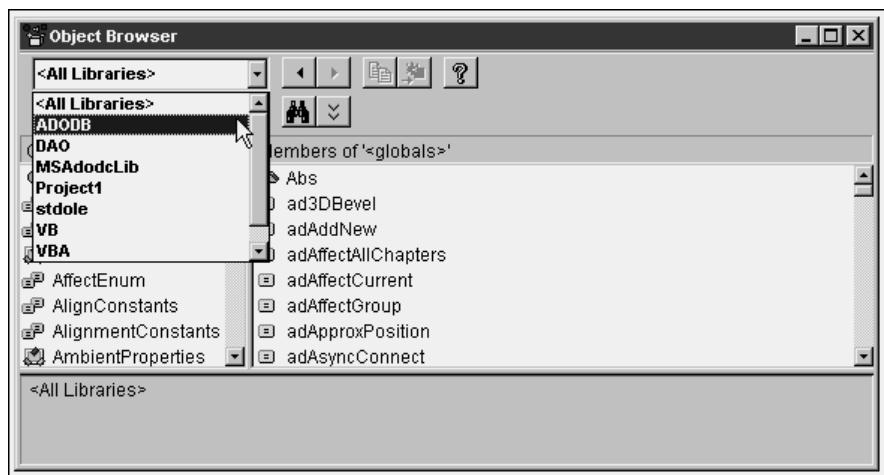
**Figure 28.17.** The selected control in the General tab of the toolbox is the Microsoft ADO Data Control.



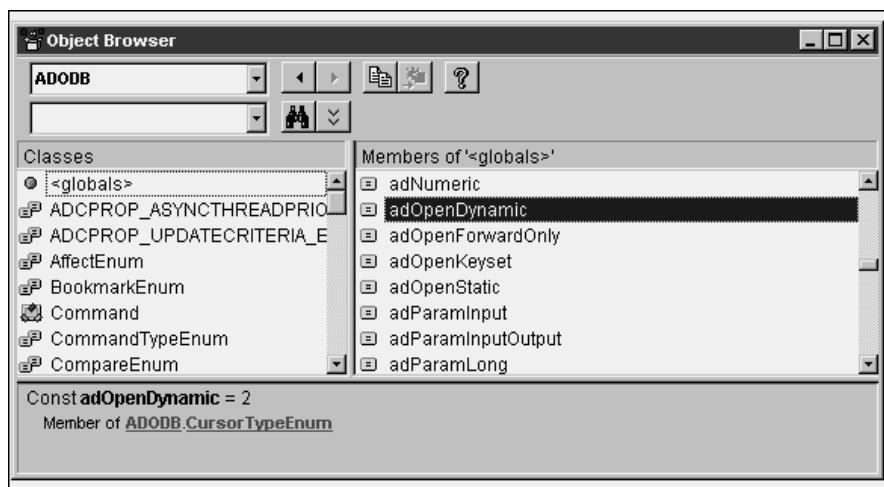
**Figure 28.18.** Placing the ADO Data Control on a Visual Basic form.

Click on the <globals> item in the Classes list box on the left, and you'll see a list of all available constants in the Members list box on the right. Scroll down to look for the constant (or constants) of interest. In **Figure 28.20**, I've highlighted the adOpenDynamic constant. In the bottom of the Object Browser, you'll see a bit of information about this member, including the declaration that sets the constant's value to 2.

This is how I found out that adOpenKeyset is equivalent to 1, for example, and how you can determine what the rest of the constants you're interested in evaluate to as well.



**Figure 28.19.** Selecting the ADODB library in the Visual Basic Object Browser.



**Figure 28.20.** The adOpenDynamic member of <globals> is a constant equivalent to the value 2.

## So should you use ADO?

Now that we're done with this whirlwind tour of ADO, I know there's still one question bugging you. Because Visual FoxPro has this great native data engine, and it's got an easy and pretty reliable mechanism to attach to back ends like SQL Server and Oracle, why bother with "yet another data access method"?

Well, if you're just dealing with straightforward data access in LAN and Client-Server applications, you shouldn't. ADO is slower than either native Fox or ODBC connections to SQL Server, and it doesn't make sense to introduce another layer of overhead just because it appears to be the popular thing to do.

But ADO has capabilities beyond these. Remember that the OLE DB architecture is predicated on a variety of disparate providers—not only providers like SQL Server, but also for Excel, Outlook, and other non-traditional, non-relational data sources. And the world of component-based development—the Windows DNA architecture—has even more requirements in terms of moving data between components.

ADO provides a method of effectively moving data between processes. You certainly don't want to build a slick, 32-bit component that can be called by the latest development tools, only to have it write out a text file of data so another component can share it. As a middle-tier tool, Visual FoxPro 6.0's COM capabilities support creating ADO recordsets and moving them back and forth to other tiers in the system.

As a result, if you're going to be using VFP as your middle-tier development tool, the marriage of VFP and ADO provides you with capabilities that you won't find anywhere else.



# Chapter 29

## Using Microsoft Transaction Server with Visual FoxPro

Unless you've been living in a cave for the past few years, you've heard of MTS—Microsoft Transaction Server—and usually in the same breath, how easy it is to integrate Visual FoxPro components with MTS. It's likely, though, that unless you've rolled up your sleeves and dug into it yourself, that you're probably a little fuzzy on the basics: What is MTS, where does it come from, and when and where would I use it in my applications? In this chapter, I'll answer these questions, and give you a foundation upon which you can explore further.

If you're building multi-tier applications, you're going to run into a number of issues that don't come up with traditional LAN or Client-Server applications. First, the specific architecture of n-tier applications dictates the use of components to handle business logic—and these components are placed between the presentation layer (the user interface) and the back-end database. The physical management of these components is a non-trivial issue—installing and updating are more complex than simply copying an executable to a user's workstation.

Second, managing transactions—blocks of processes that may be distributed across more than one component—requires mechanisms that are beyond the scope of any single development tool, such as Visual FoxPro or Visual Basic. And, finally, one benefit of multi-tier applications can also turn into an Achilles heel—the deployment of an application across the Web implicitly offers availability to virtually an unknown number of users. Suddenly, scalability becomes a major issue, as the potential pool of users can't be controlled as easily as it could be for a LAN or Client-Server application.

Visual FoxPro and Microsoft Transaction Server are a great combination for dealing with these issues. Visual FoxPro, with the inclusion of new capabilities in Service Pack 3, allows you to create components that can be accessed by multiple users at the same time with little performance degradation. Microsoft Transaction Server allows you to combine components into "packages" that can act as a single transaction, and provides a friendly user interface to deploy those components. Together, they provide the mechanism for managing the middle tier of a multi-tier application.

### What is MTS and where does it come from?

Microsoft Transaction Server is a component-based transaction processing system for building, deploying, and administering robust Internet and intranet server applications. Well, that's what the marketing literature says. To you and me, it's a piece of software that comes with the NT Option Pack for Windows NT Server, and is used for creating and administering groups ("packages") of components that are used in multi-tier applications. No big mystery after all, really.

In addition, MTS provides role-based security, connection pooling, and multi-database transaction support.

Role-based security provides another layer of security, in addition to the group and user levels available in Windows NT. Each interface—each component—can have a different level of security defined, which provides for great flexibility and power.

Connection pooling is the ability for multiple users to share a single connection. MTS automatically adjusts the number of connections open for users so as to balance demand. Without connection pooling, 500 simultaneous users would require 500 connections—and thus a box with about 5 terabytes of RAM (well, maybe not 5 terabytes, but an awful lot) in order to have enough resources for those connections. With connection pooling, those 500 users would only need, say, 25 or 50 connections—a considerably less expensive option.

Most databases provide for some sort of transaction handling these days. Thus, a series of processes, say, several adds and edits, can all be packaged into a single transaction, and the entire batch can be committed (or canceled: “rolled back”) as a single unit. And, as I mentioned, MTS allows you to package multiple components into a single transaction as well. However, what about transactions that have to span more than one database? What if you have to manipulate data in more than one database, but want all of the manipulations to be committed or rolled back as a single unit? Most databases don’t provide for this heightened level of transaction handling. MTS will allow you to do this—span transactions across multiple databases.

## Installing and accessing MTS

You may very well have MTS already installed on your server without knowing it. When you install the NT Option Pack on an Windows NT Server machine, it will allow you to select a number of choices, including Microsoft Message Queue Server, Internet Connection Services for Microsoft Remote Access Server, Internet Information Server and Microsoft Transaction Server.



*The NT Option Pack is available in many places. For example, it’s included with the Enterprise Edition of MSDN, with new versions of Windows NT, and can be downloaded (all 27 MB of it) from the Microsoft Web site at [www.microsoft.com/windows/downloads/default.asp](http://www.microsoft.com/windows/downloads/default.asp).*

If you don’t have it installed, just grab the Option Pack and run the installation like you would any other standard Windows setup—and be sure to select MTS as one of the components to install.

Once you’ve got it installed, you get to navigate through the maze of submenus to access it. From the Start menu, select Programs, Windows NT 4.0 Option Pack, Microsoft Transaction Server, Transaction Server Explorer, and you’ll wind up with a dialog that looks something like **Figure 29.1**.

As you drill down through the tree view in the left pane of the explorer, you’ll see several nodes of interest. The first is the My Computer node, which makes sense if you’re running this from your own NT Server box.

Underneath My Computer you’ll see the Packages Installed node, which is shown in exploded view in **Figure 29.2**. Now examine what the node for a package looks like. You can right-click on a specific package to open a context menu for that package. Selecting the

Properties menu option opens the Properties dialog for that specific package, as shown in Figure 29.2.

You can also examine the methods available in a specific component—remember, that's the whole point of a component, to provide functionality to other parts of the application through the exposure of public methods. Select the Methods node under a specific Interface node, as shown in Figure 29.3.

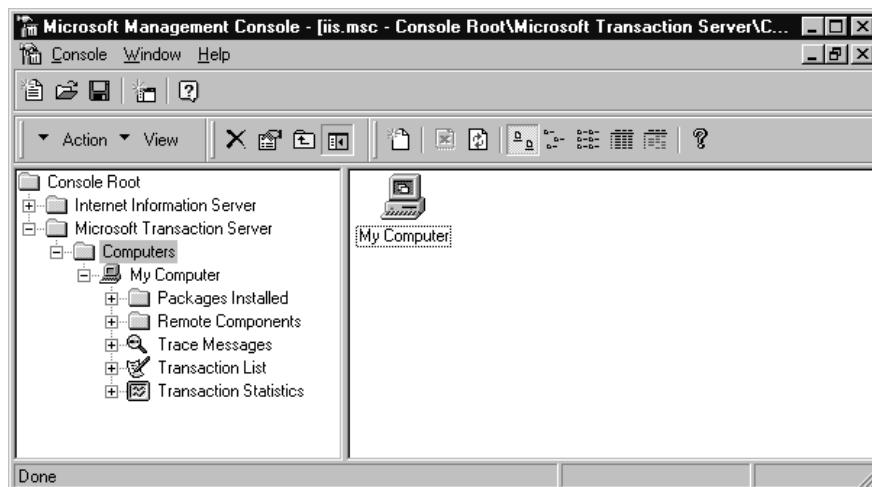
## Creating a package

A package is a group of one or more components that you want to treat as a single cohesive unit. To create a package, right-click on the Packages Installed node and select the New, Package menu option from the resulting context menu, as shown in Figure 29.4.

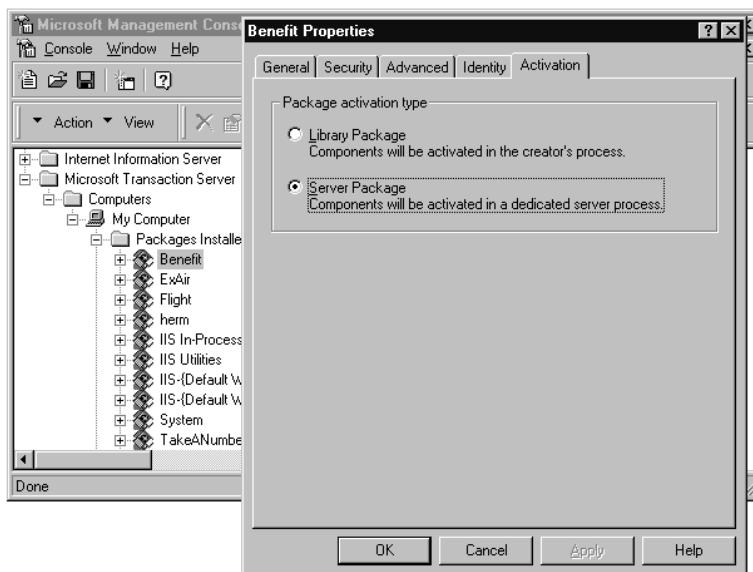
The Package Wizard will launch as shown in Figure 29.5, and will give you the option of installing a pre-built package into MTS, or creating an empty package. If you create an empty package, you'll need to add components to it.

Since you probably don't already have pre-built packages while you're reading this chapter, I'll head down the "Create an empty package" route. The first step is to enter a name, as shown in Figure 29.6. As you can see, MTS is still in its infancy, as evidenced by the rudimentary dialogs.

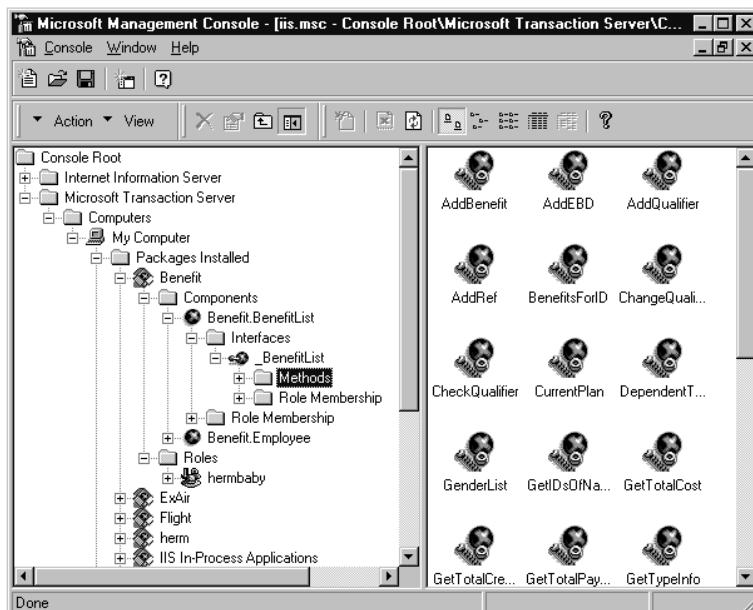
Next, you'll attach the package to a specific user account, as shown in Figure 29.7. If you're logged on as the user who will eventually be using the package, you can select the default; otherwise, click the "This user" option button and the Browse button to select another user. Typically you'll have a user on the server who is set up for access to this application.



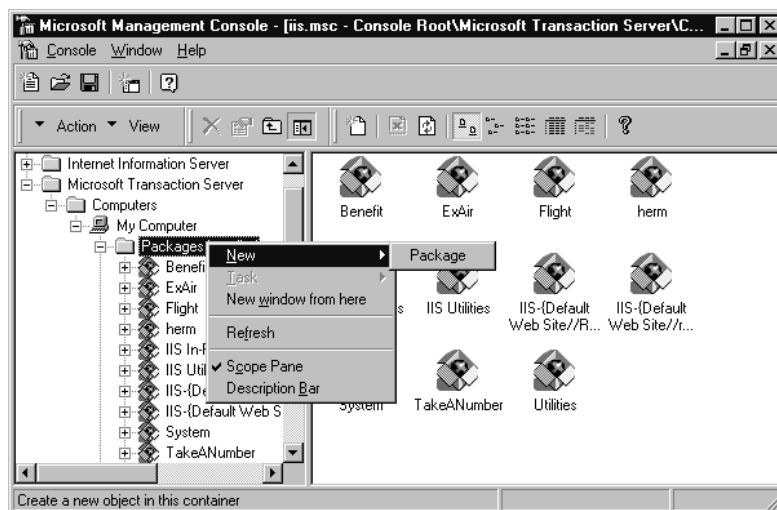
**Figure 29.1.** The Microsoft Management Console sports an Explorer-style interface that allows you access to MTS, as well as other parts of the NT Option Pack.



**Figure 29.2.** The Properties dialog for a package allows you to set a variety of properties, including security and how the package will be activated.



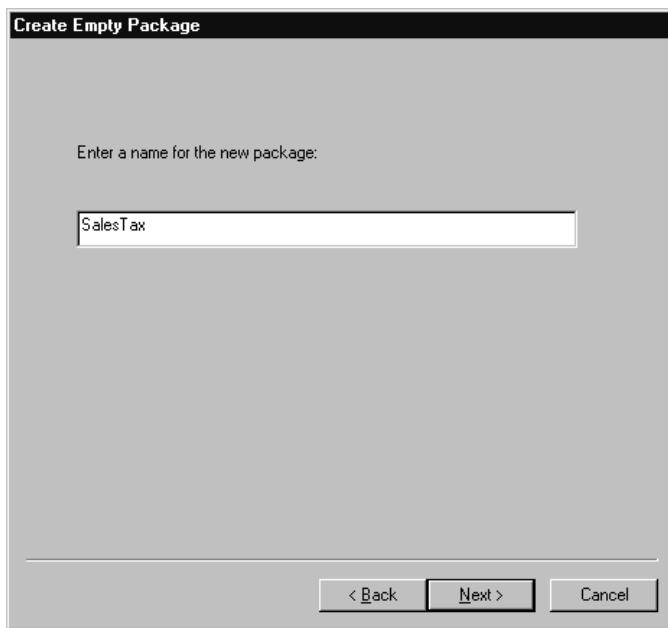
**Figure 29.3.** The Methods node under a component's Interface displays all public methods for that interface.



**Figure 29.4.** Right-clicking on the Packages Installed node in the MTS Explorer allows you to add new packages.



**Figure 29.5.** The Package Wizard allows you to install pre-built packages or create a brand new package that you can add your own components to.



**Figure 29.6.** The first step in the Package Wizard is to enter a name for the package you are creating.

Once you're done creating your package, you've got an empty package. When you look in the MTS Explorer, you'll see your new package, with a pair of nodes—Components and Roles—underneath it. However, much like any other box without anything in it, your new package isn't much use until you put something in it. When you right-click on the component node under your package, you'll be able to select the New, Component menu option from the context menu, as shown in **Figure 29.8**.

The New, Component menu option will launch the Component Wizard, as shown in **Figure 29.9**.

If you choose to install new components, you'll be presented with the Install Components dialog as shown in **Figure 29.10**. You can choose between .DLLs, .TLBs, and all files in the file filter.

Once you select the desired file (you *do* know which file contains your components, don't you?), the components contained in it will be listed in the Components found list box.

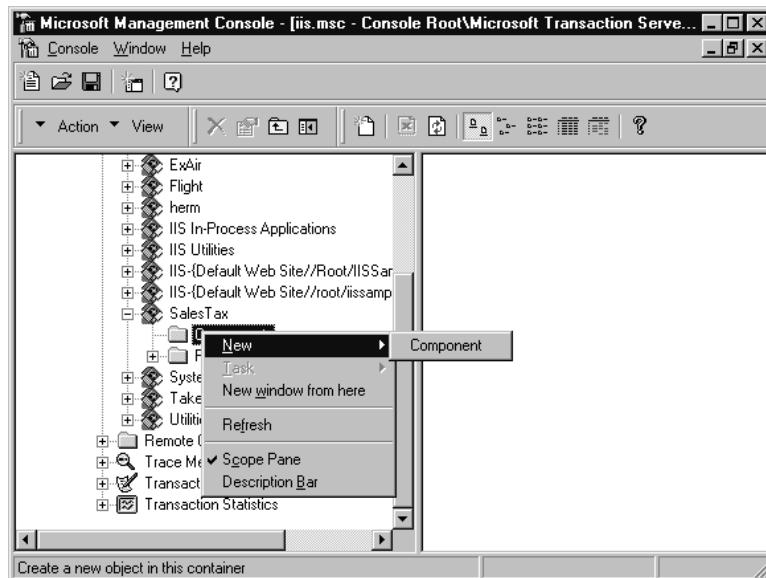
If you select the “Import components that are already registered” option, you'll get the screen displayed in **Figure 29.11**. This dialog is a perfect example of why we all have jobs, and will continue to have jobs even after programming is reduced to “assembling components into systems.”

Once you're done with the Component Wizard, the components you installed will be listed in the Components node under your package.

You can also add Roles to your Package by right-clicking on the Roles node and selecting the New, Roles menu option, as shown in **Figure 29.12**.



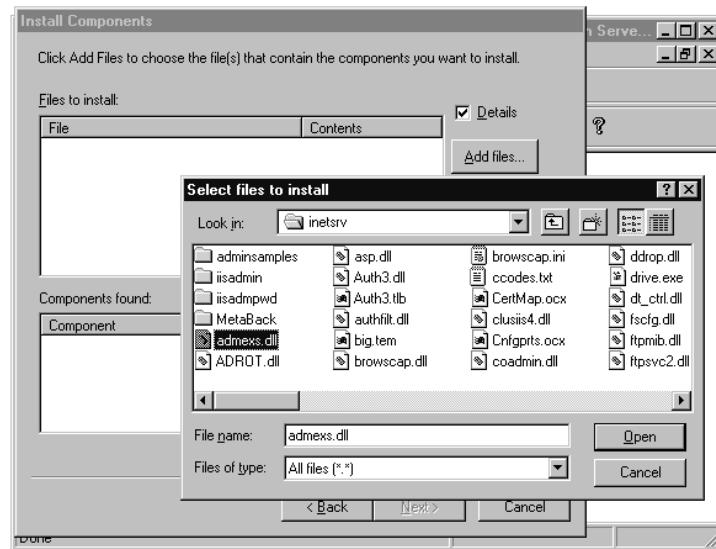
**Figure 29.7.** The next step in the Package Wizard is to set the Package Identity.



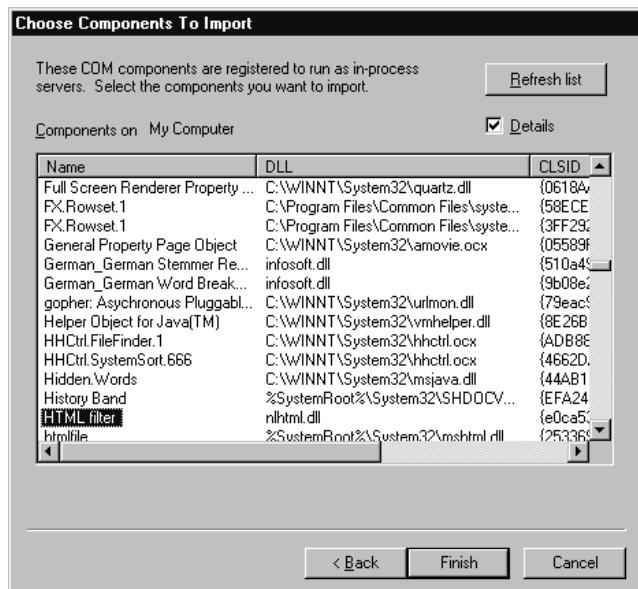
**Figure 29.8.** Use the Component context menu's New, Component menu option to add components to a package.



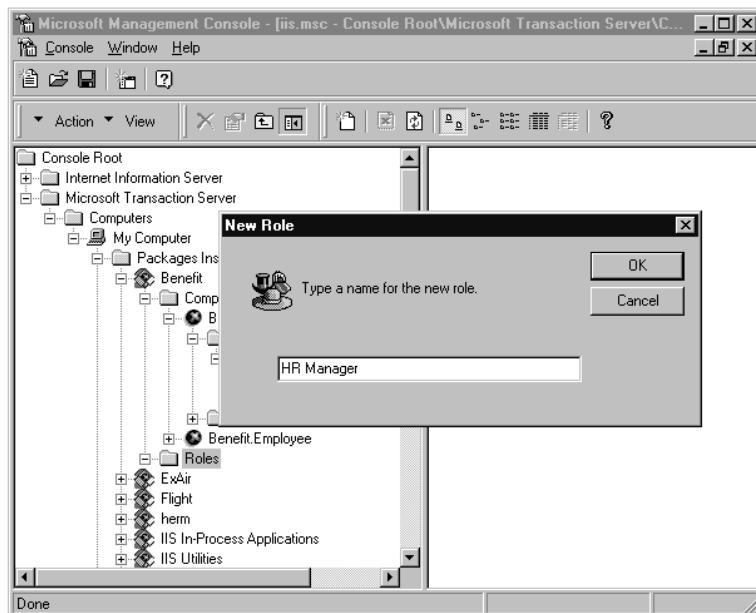
**Figure 29.9.** The Component Wizard allows you to install new components or add components that are already registered with Windows.



**Figure 29.10.** Clicking the Add files button in the Install Components dialog allows you to navigate through the directories to select the file that contains the component(s) you want to install.



**Figure 29.11.** You can select components that have already been registered on the local machine.



**Figure 29.12.** Right-clicking on the Roles node in the MTS Explorer allows you to add new roles to a package.

## More information

So now that you have an idea about how to create a package and add components to it, it's time to talk about a couple of Visual FoxPro-specific issues, and where to go for more information. VFP 6.0 provided the ability to create COM components that work in MTS. However, until Service Pack 3, VFP COM components were somewhat limited in their utility, because they exhibited a behavior known as "method blocking."

In short, two requests to the same server would queue up in serial order, with the first request blocking the second one from executing while the first one was still executing. One of the nifty features of SP3 was a new multi-threaded runtime that eliminated this problem. If you examine the WINNT\SYSTEM32 directory, you'll see two runtime files, VFP 6R.DLL and VFP6T.DLL. The first is the normal runtime that is distributed for most applications. The second is a special multi-threaded runtime that you'll use when creating in-process servers that require high scalability.

These issues are discussed in detail in the SP3 online help file, VFPSP3.CHM, found in the Program Files\Microsoft Visual Studio\VFP98 directory.

You'll also want to learn more about implementing security, and how to handle transactions and deploy packages. Randy Brown has written an excellent white paper for MTS and Visual FoxPro that covers these topics and more. It can be found at the Visual FoxPro Web site at [msdn.microsoft.com/vfoxpro](http://msdn.microsoft.com/vfoxpro). Select the Technical Resources link on the left side of the page, then the Technical Articles link, and click on the COM and ActiveX Development topic. The name of the paper is, interestingly enough, "Microsoft Transaction Server for Visual FoxPro Developers."

# Chapter 30

## A Visual Basic Primer for Visual FoxPro Developers

**Let's face it. As you saw from Chapter 28 on ADO, you're going to have to get acquainted with Visual Basic one of these days. In this chapter, I'm going tackle VB from a Fox developer's point of view—pointing out similarities and differences, what to look at, what to ignore. And, most importantly, I'm going to provide a very fast path to getting up to speed, because I know what your background is. Unlike those general-purpose books, I don't have to write for a wide range of developers in the audience.**

**Much of the material for this chapter came from articles originally published in *FoxTalk*. For more information, see [www.pinpub.com/foxpro](http://www.pinpub.com/foxpro).**

You picked up this book because it's about Visual FoxPro, not Visual Basic. There are a thousand VB books out there, and you're a Fox developer. So you might be taking issue with my hypothesis in the introduction to this chapter. So let me explain.

You're going to need to become familiar with VB for three reasons. But those reasons aren't the ones that you might be thinking of.

It's not because "they're going to kill Visual FoxPro." The next release of Visual FoxPro is already floating around in certain circles, and if you subscribe to *FoxTalk*, you'll be reading about it shortly.

It's not because Visual Basic is "better" than Visual FoxPro. It's a general-purpose programming language with its own strengths and weaknesses. Fox is a data-centric programming language, again with its strengths and weaknesses. Each does some things well, and other things, er, poorly. "Better" is truly in the eyes of the developer.

And it's not because I've gone bonkers. (Be charitable, please!)

The first of the three reasons is that you might think I'm lying about the future of VFP, and figure it's safe to cover your bases. Well, I'm not lying—just like any dutiful Microsoft shill, I'm simply repeating what I'm told to. Okay, seriously, not putting all your eggs in the same basket is probably a good idea, regardless. Ever really wonder why we're still saddled with a Report Writer and Menu Builder from 1992, and there are no plans for updates? Maybe it is time to look elsewhere for those components!

Second, "everybody" has it. And you want to be popular, don't you? How many of you preferred another word processor or spreadsheet, but gave up because every document you received was in Word 6.0 format and you got tired of converting—and having pieces lost in the process? It was just easier to go with the tide; besides, you probably got a copy of it free from somewhere or another.

As a result of everyone having VB, when you see sample source code, it's often in VB (or VBA). Wouldn't it be nice if you were more familiar with the language so you understood what DIMs and VARIANTs were, instead of having to guess?

The last reason is that, as alluded to earlier, there are some things that Visual FoxPro just doesn't do very well. For example, I've been wrestling with the integration of a high-end imaging ActiveX control in VFP. It's been an ugly road, and as I'm writing this, it looks like the road is a dead end. The customer isn't really very interested in hearing about "the differences in VFP's and VB's containership and hosting abilities." (There's the one guy in their IS department who's muttering, "I knew we should have written this in VB.") They just want it to work.

Why spend hours and days working around a weakness when another tool can be used that doesn't have that weakness? How about, as Dan Freeman suggested, "building an OCX in VB? Just wrap it around the recalcitrant control. The control will think it's hosted in VB and Fox seems to work better with VB controls."

Well, if you go that route, and I think it's one worth considering, you'll need to get up to speed with VB. While that's not a significant challenge (hey, there are only something like 19 keywords in the entire language), it could still be a longer process than you have time for. You could pick up one of those "Learn VB in 21 Days" books—you know, one that sits next to the 2000-page "Write your own NT kernel in 24 hours" book. (I swear—I saw one of these books!) But those are written to be as generic as possible, aimed at everyone from the experienced VB5 developer to the novice who still hasn't gotten the hang of his mouse. We can do this in a more expedient fashion.

## **The Visual Basic IDE**

I'm assuming you've installed Visual Basic 6.0 that comes with the Visual Studio 6.0 package, and for the time being, it doesn't matter if you've got the Professional or Enterprise version. You can also get VB in some other versions—Training, Standard, and Mega-Galactica—but I'll ignore those. When you load VB, it will present you with a dialog that asks you what type of project you want to create. I checked the "Don't show this dialog ever, ever, ever, ever again" check box because when you create a new project from the File, New Project menu command, you'll get access to the same dialog.

### **MDI versus SDI**

Once VB loads, you'll see an MDI window filled with windows, much like Visual InterDev. See **Figure 30.1**. This might startle some of you who have seen previous versions of VB—those that used an SDI interface, which was rather disconcerting to us Fox developers. The windows just sat on the desktop and you could see the desktop showing through the holes. (You can turn SDI on through the Tools, Options, Advanced tab.) See **Figure 30.2**.

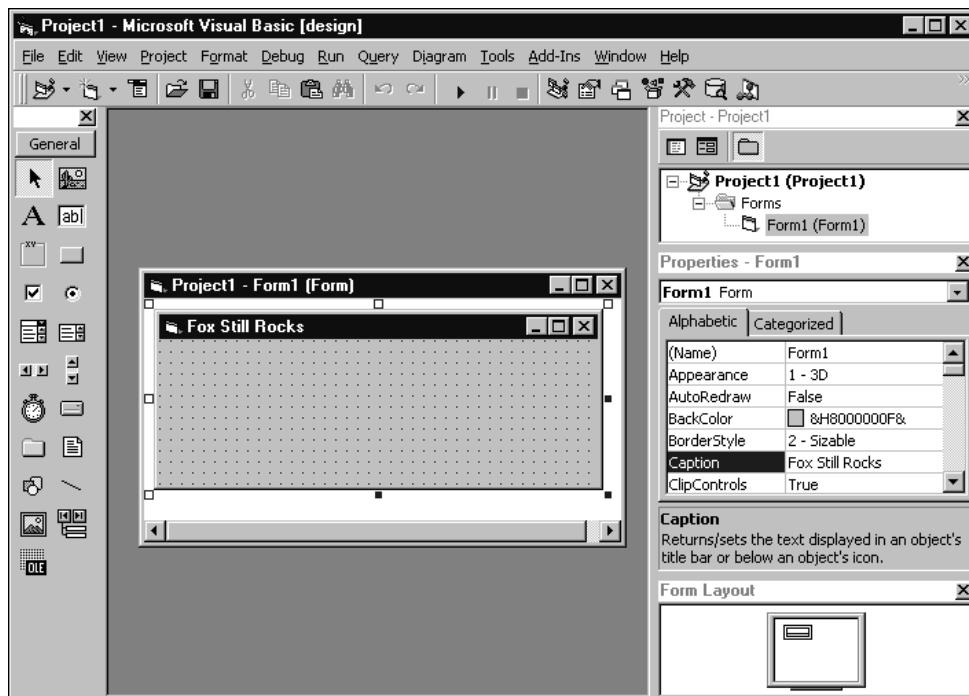
### **What are all these windows?**

What are all of these windows? On the left side is a long, narrow window with about a dozen controls in it—that's your Controls toolbar, just like Fox. If you let your mouse icon hover over a control, you'll get a ToolTip describing the control, just like every other Microsoft product on the planet. (I was going to say "Windows product" but figured it's pretty close to the same thing.) The bar across the top that is captioned "General" is called a tab, even though you and I think it looks like a "bar." However, the toolbar can contain lots of controls—and I mean lots. The native controls that come with VBE (Visual Basic Enterprise) number close to 125.

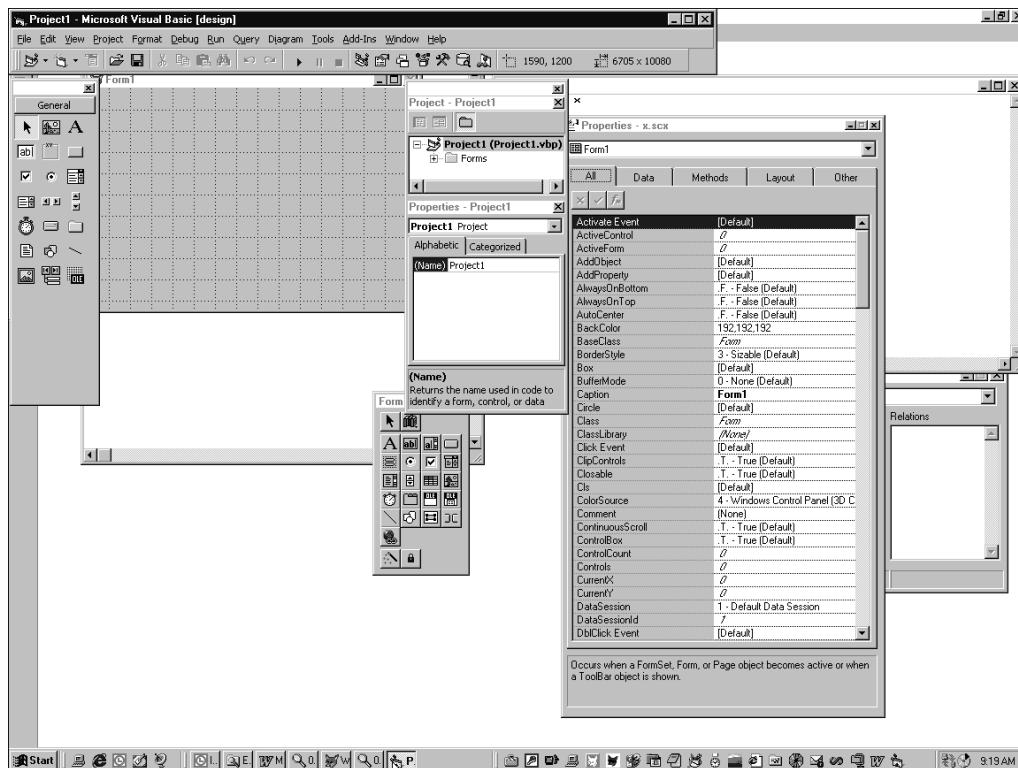
However, instead of segregating controls into different toolbars like Fox does (Standard, ActiveX, and your own libraries), you can separate the Controls toolbar by adding tabs to the toolbar, each of which will contain a certain type. See **Figure 30.3**.

VB uses a project metaphor to build applications just like Fox, and is heavily dependent on forms, again, just like Fox. When you create a new project (either through the New Project startup dialog or the File, New Project menu command), you'll be faced with a plethora of options. Pick Standard EXE for the time being, and you'll get a project named Project1 in the upper right hand window that originally didn't have a caption in the title bar. This, then, is the Project Explorer.

Underneath the Project Explorer is the Properties window, which provides the same function as our own Properties window, although the functionality is a bit different. You'll already see that you can sort the properties alphabetically or by category—Appearance, Behavior, Font, Misc, and so on. I've kept the properties sorted alphabetically because it's hard enough keeping track of which properties belong to VFP and which to VB, and what the minute differences are. Trying to remember if "Visible" is an Appearance property or a Behavior property is just too hard. You'll see I've already changed the caption of Form1 to something more aesthetically pleasing—and I didn't even have to read the manual to find out how!



**Figure 30.1.** VB 6.0 sports an MDI interface by default.



**Figure 30.2.** Use the Tools, Options Advanced tab to change VB 6.0's interface to SDI—if you can stomach the mess!

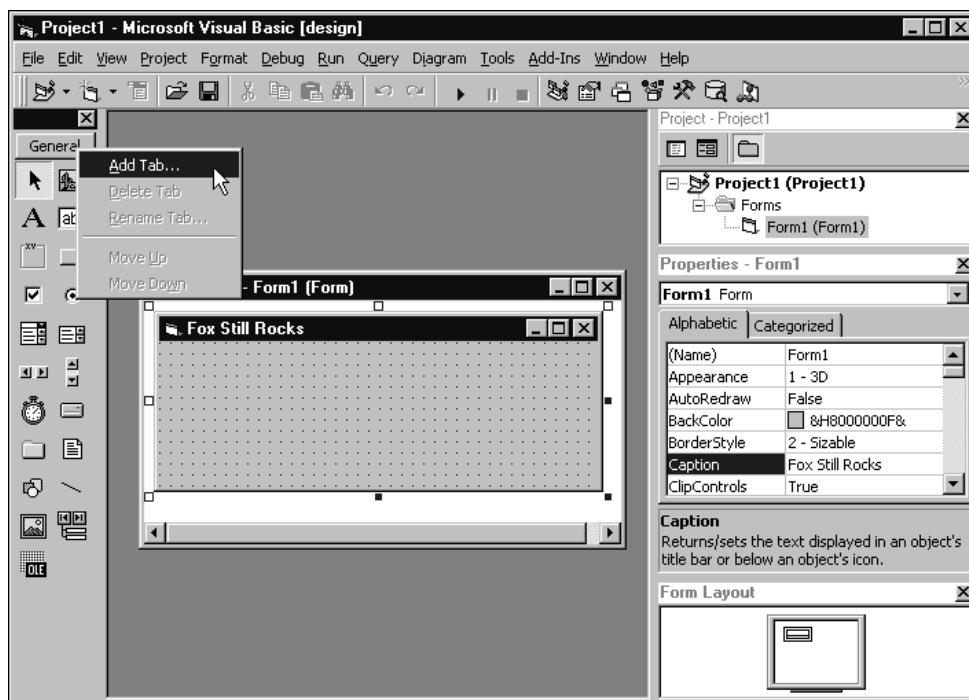
New to you is the Form Layout window in the lower right, which shows you a bird's-eye view of what the Form will look like on the desktop. As you resize a form, you'll see the corresponding view in the Form Layout window change as well.

## Your first VB form

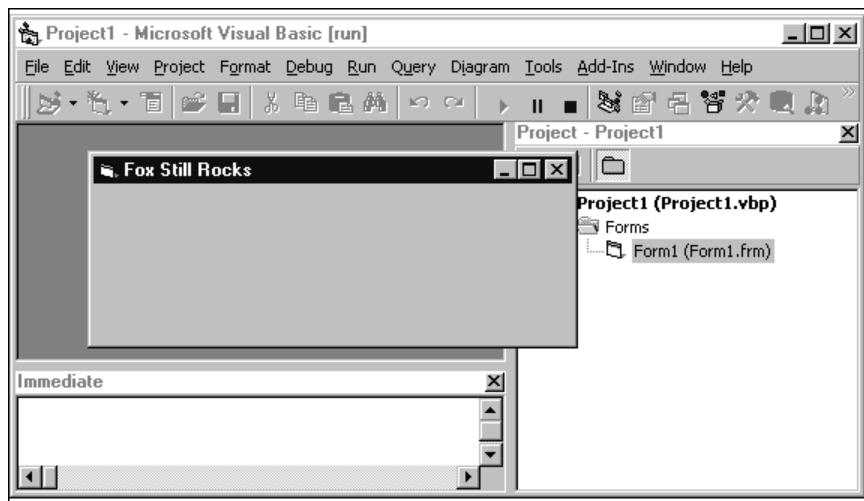
The form that is created when you create a project is named “Form1” by default. By itself, it isn’t very interesting, but you can run it just as you can run an empty VFP form. To do so, you can click the Start button in the Standard VB toolbar (it’s the button with the arrowhead pointing to the right, under the “m” in the Diagram menu if you’ve got both the menu and Standard toolbar docked and nudged up to the left). Alternatively, you can issue the Run, Start menu command or simply press F5. See **Figure 30.4**.

You know what’s cool? If you task-switch with Alt+Tab, you’ll see that the VB form is not running inside like VFP—it’s an SDI form automatically.

You might be thinking, “Okay, how do I stop this thing?” You can click the form’s close box, or press the ever-so-intuitive Alt+F4, or just click on the toolbar button with the square box (two buttons to the right of the Start toolbar button).



**Figure 30.3.** Right-click on the General tab in the Controls toolbar to add, edit, delete, and manipulate tabs.



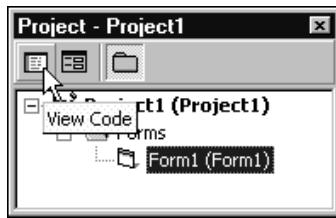
**Figure 30.4.** The Fox Still Rocks form in, er, action.

## Adding code to a form

I know that example is less involved than the typical “Hello World” program you first write in C, but, hey, this is BASIC. Let’s put some code in this form.

Code windows are one significant departure in behavior from “the Fox way.” First, in order to get to them, you can’t use the Properties window. It’s for properties, after all. It doesn’t say “Properties and Code Window,” does it?

To open the Code window, you again have nine thousand ways to do so. You can highlight an object in the Project Explorer and click the View Code button in the project, as shown in **Figure 30.5**.



**Figure 30.5.** The left button in the Project Explorer toolbar opens a code window.

You can also issue the View, Code menu command, but the easiest way is to just double-click the form. After any of these operations, the Code window displays as shown in **Figure 30.6**. The left drop-down shows the various objects of the form, including “General” (I’ll get to that later), “Form”, and every control on the form. The right drop-down contains all of the events into which you can stuff code—and you’re familiar with many of these, such as Activate, Load, Resize, and Unload.

In Figure 30.6, I got ahead of myself and started writing some code. Just when you got used to typing “MessageBox” in VFP, VB comes along and uses “MsgBox” to perform the same function. You’ll notice in Figure 30.6 that VB has “Code Completion”—a ToolTip will appear, prompting you about the parameters that can be used with the function you’re typing.

You’ll also see that code for a particular procedure is bracketed by “Private Sub <name>” and “End Sub” statements, just like we use FUNC and ENDFUNC in Fox. “Sub” is a holdover from the olden days when programmers used to write “subroutines,” and VB puts them in for you automatically. If you delete one of those lines, “unexpected results may occur.”

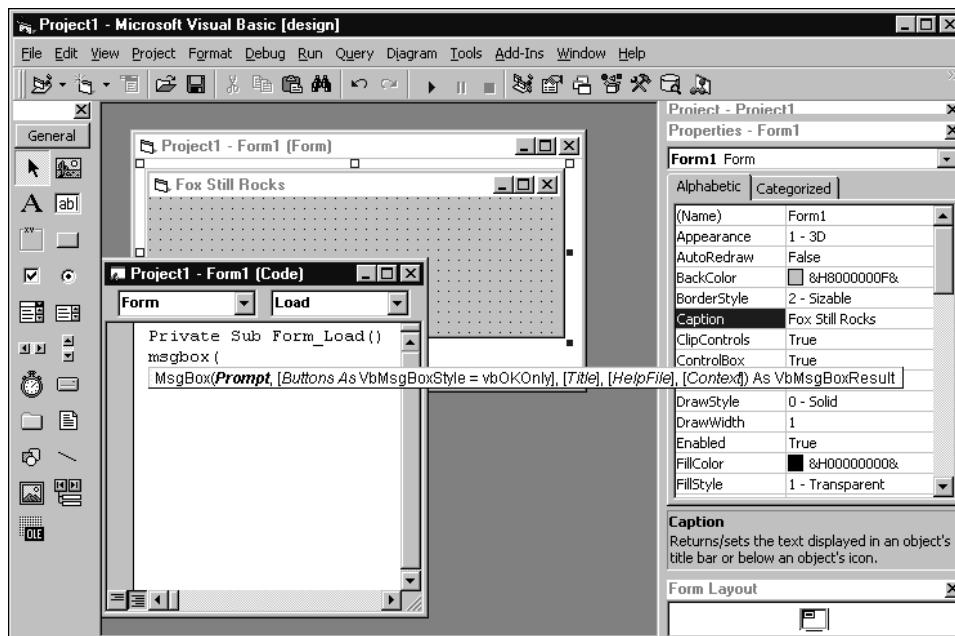
I suppose I could have put a message box in the form’s Click() method, but that would have been going from the excruciatingly trivial to the nearly excruciatingly trivial. Instead, I added a command button from the Controls toolbar, double-clicked it, and entered the MsgBox in there. Obviously, running the form will generate a form with a command button on it, and clicking on it will display a message box.

## The VB Code window

But let’s talk about the Code window for a minute because there are some new things showing up. In **Figure 30.7**, I’ve shown the Code window with pieces from several methods. As you can

imagine, this is pretty handy—being able to see code for more than one procedure in the same window (as long as those procedures are small).

Notice two buttons to the left of the horizontal scrollbar in the bottom of the Code window. The right button is selected in Figure 30.7, and indicates that you want to see more than one procedure at a time. This is the Method View button. The left button, Procedure View, limits the display to one procedure at a time.



**Figure 30.6.** The VB Code window with a command in Code Completion mode.

## Saving and printing projects

A project is a file on disk, just like in Fox, but data in it isn't automatically saved like that in a .PJP, so you have to do it yourself. However, when you save a project, it will prompt you to save all of the unsaved components, and if they're unnamed, you'll be prompted to name them along the way.

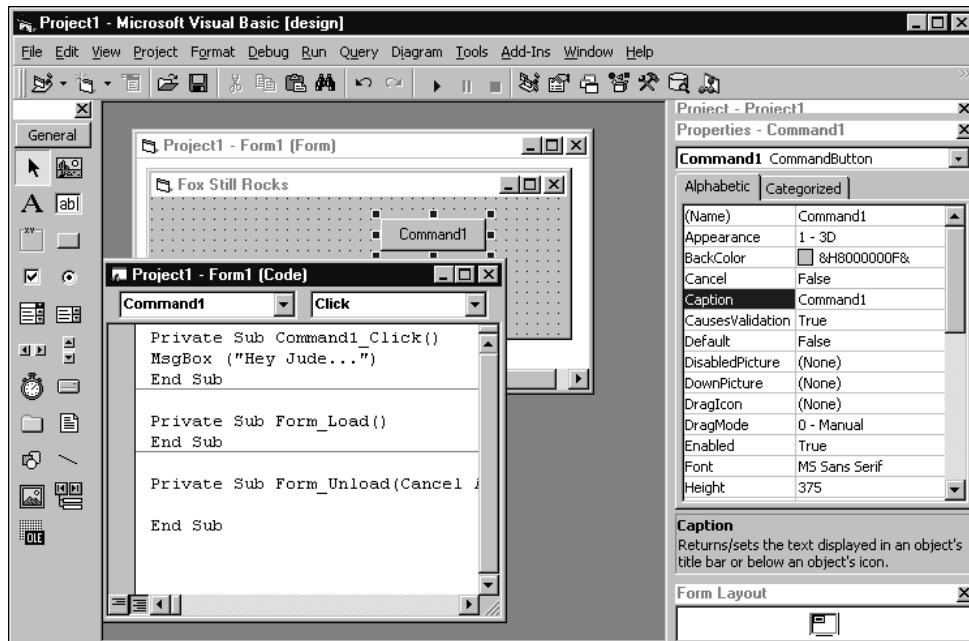
You can also print the contents of a project by simply selecting the File, Print menu. The Print dialog contains a number of options that correspond directly to VB projects. See **Figure 30.8**.

## Project components and files

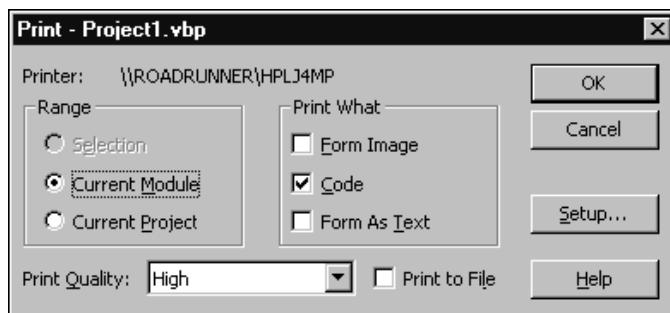
I also should mention what the structure of a project looks like and what the files on disk are. The project is stored with a .VBP extension, and is simply a text file. A VB form is also just a text file, with an extension of .FRM. I've shown both of these opened in Notepad windows in **Figure 30.9**. A third type of component that I haven't discussed yet, but that you've probably

seen here or there, is the Module. It's got an extension of .BAS, and contains code that isn't tied to a form—much like a separate FoxPro procedure file.

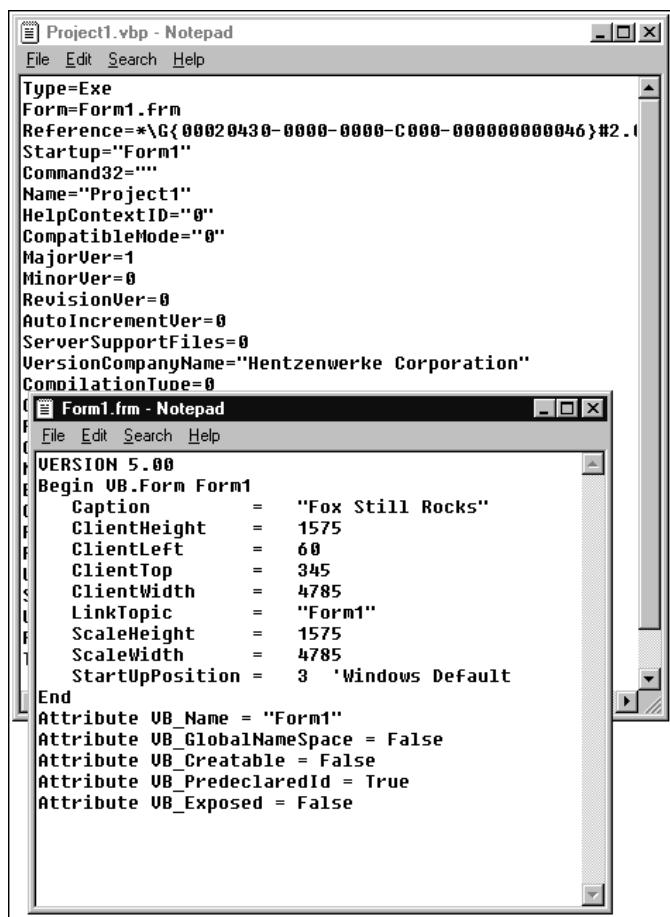
There you go. I've been through six or eight VB 6.0 books, and they've each taken 40 to 50 pages to cover this material. You now know more than what readers of those books do, and you're probably not even done with that can of Jolt. Not bad, eh? Repeat after me: Thank you, thank you, Sam-I-Am, I like VB6, Sam-I-Am!



**Figure 30.7.** Click the Method View button next to the horizontal scrollbar in a Code window to see all of the procedures in the same window.



**Figure 30.8.** You can customize what parts of a VB project should print.



**Figure 30.9.** VB projects and forms are simply text files.

I've been pretty flip about the use of some terminology—events, methods, procedures, code snippets, windows, and so on. It's time to get more rigorous.

 You know how there are certain behaviors in VFP that just don't seem right? The ones that make you ask yourself, "What were they thinking?" And when the explanation offered is "It's that way by design," you just roll your eyes. Take heart—VB has its share of incredulous behaviors.

**Stupid VB trick #1:** Add a command button to the form, double-click to open the code snippet for that button's Click event, and enter some code. Then delete the button. The code snippet stays put—it's not deleted.

**Stupid VB trick #2:** Add a command button to a form. Double-click to open the code snippet for that button's Click event, and enter some code. Change the name of the button from "Command1" to "MyButton." The code snippet for "Command1" stays there—with the code you entered. Meanwhile, running the form and clicking MyButton won't do anything, because the code is still attached to Command1 (even though it's nonexistent), not MyButton.

**Stupid VB trick #3:** Repeat the steps in #2, but then add another command button. It will be named Command1—and the previously orphaned code will now be attached to the new button!

## The basics of writing code

### The Properties window, property pages, and toolbars

If you open the VB Properties window, you'll see a drop-down list box just like in VFP—it's called the "object box" and, like VFP, contains a list of all objects in the currently selected form. However, if you open it, you'll notice that all of the objects on the form, as well as the form itself, are listed in alphabetical order—and there's no apparent hierarchy involved. I'll discuss this in more detail in the next section.

The Properties window works like VFP's: Select an object, and the associated properties and their values appear in the list box below. The two tabs above the list box allow you to choose whether you want to see the properties in alphabetic order or by category.

If you select the Categorized tab, you'll see a third column appear on the left side of the list box—a series of plus and/or minus signs that allow you to expand or contract the list of properties for that category. I'll probably get around to describing a bunch of "tips and tricks" for various control properties later, but here's one that you'll likely want right away. In VFP, you use the "\<" character string in a command button's caption to set the next character to act as a hotkey; in VB, you use the ampersand (&) character. Thus, the VB would use the text string "&Done" to accomplish the same task as the VFP caption "\<Done".

Another difference between VB and VFP is that while right-clicking on a form displays a context menu with a Properties menu command, and selecting the Properties menu command will open the Properties window, right-clicking on a some controls in a form and selecting the Properties menu command will bring up a "property page." This property page contains a tabbed dialog that contains various physical or visible attributes of the control. More on this later.

Visual Basic's control toolbar is called the "Toolbox", and you can customize it by adding non-default Microsoft, or third-party controls to it. First, select (or create) the tab (the gray object that you and I would have called a "bar") under which you want to add the control. (Remember that you can add a new tab by right-clicking in the Toolbox and selecting the Add Tab menu command.) Next, right-click under the tab, select the Components menu command, and use the resulting dialog to select the control you want to show on the toolbox.

Finally, remember that the Properties window shows only properties—not methods or events. I'll discuss those shortly when I get to programming.

## VB's version of containership

Earlier I mentioned that there is no hierarchical display of objects in the Properties window object box. This is because VB doesn't have the same type of containership as VFP—and while confusing, this actually makes life easier in some cases. For instance, if you put a check box on page 2 of a page frame in VFP, you use the following code to reference its caption:

```
thisform.pageframe1.page2.checkbox.caption
```

To reference the same check box's caption in VB, you simply use the code:

```
check2.caption
```

This tells us a couple things. First of all, you will wear down your keyboard less, since you have to do much less typing. Second, every control on a form has to have its own unique name—regardless of where it lives. Thus, while you can have a “checkbox1” on each page of a four-page tabbed dialog, you'd have “check1”, “check2”, “check3”, and “check4” in VB. Third, the containership, or lack thereof, is going to have some serious ramifications on programming that I'll discuss later.

## Slinging code

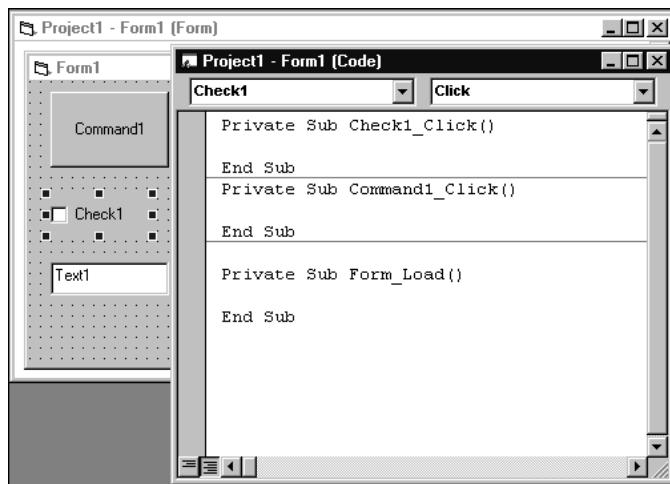
As I just mentioned, when you write code, you don't have to use a fully qualified name for a control—simply the name of the control and its property or method name. Second, in order to get to the code, you can either click the View Code button or just double-click on the form. But here's where it can get confusing. Remember that you can use the icon in the bottom left of the window to display all the code for the form, or just the code for the current object (either the form, or a control on the form). When you double-click a control that has no code attached to it, you'll get a new module for the Click() event of that control already built for you, as shown in **Figure 30.10**. Otherwise, double-clicking a control with code in one of its events will bring up the code for that event.

The Code window can contain code for one or more events, for one or more controls. Each “chunk of code” attached to an event has two parts. The first is the declaration of the module, and the second is the termination of the module, like so:

```
Private Sub Form_load()  
<your code goes here>  
End Sub
```

Each of these “chunks of code” is referred to as a subprocedure, or a function, or, generically, a code segment. The “Sub” keyword means that this is a subprocedure—much like procedures in Fox. (I'll discuss the difference between procedures and functions shortly.) The “Private” keyword means that this subprocedure scope is the form—it can be called by code elsewhere in this form, but not anywhere else in the project.

The subprocedure's name consists of the name of the control and the event to which the module is attached. But here's a warning: Putting code in a module and then changing the control's name will reflect in the module being orphaned.



**Figure 30.10.** Double-clicking a control formerly without code opens a new module for the Click() event of that control.

The Code window also has a section called “General.” (It’s the first entry in the object drop-down at the top of the Code window.) You can use this area to create your own subprocedures and functions, much like form-level methods in Fox.

The parentheses following the name of the control enable you to pass parameters to the code segment. For example, suppose you have a form with a check box and a command button on it. You could use the following code to pass the caption of the control to a form-level function that would display the caption whenever the control was clicked:

```

Private Sub MySub(xmessage)
  MsgBox (xmessage)
End Sub

Private Sub Check1_Click()
MySub (Check1.Caption)
End Sub

Private Sub Command1_Click()
MySub (Command1.Caption)
End Sub

```

In this brief example, you see that the same basic programming principles apply—calling a subroutine with a parameter. And because MySub is declared as private, you can’t go off and call that very valuable subprocedure from another component in the project.

## Procedures vs. functions

The difference between subprocedures and functions in Visual Basic is the same as in VFP—subprocedures don’t return values; functions do. You know the rest. Well, actually, you don’t

know all the rest. You define a function (a user-defined function, of course—VB also comes with its own set of functions that work just like in any other language) like so:

```
Private Function AlphabetizeName(FirstName, LastName)
AlphabetizeName = LastName & " " & FirstName
End Function

Private Sub Command1_Click()
    MsgBox (AlphabetizeName(Check1.Caption, Check4.Caption))
End Sub
```

The AlphabetizeName function takes two parameters as input, reverses them, and concatenates them with a space between. The Command1 button takes the captions of two check boxes on the form and passes them to the AlphabetizeName function. The result of the AlphabetizeName function is then sent as a parm to a MsgBox function as the message to display.

The most interesting thing to note about VB functions is how to return values. The AlphabetizeName function has a variable with the same name as the function that receives the value to be returned to the caller. Once you've got that down, the rest is easy.

## Option Explicit

Visual FoxPro is a “weakly typed” language, which means that you can assign any old value to any old variable, and then change the value of the variable any old time you want. This provides terrific flexibility, but at the same time exposes you to untold amounts of danger, which you are well aware of.

Visual Basic has this same capability—weakly typed variables—but you can also force VB to make you define what variables you are going to use, and what data type they will be. This is called “strong typing” and is done by checking the Require Variable Declaration check box in the Editor tab of the Tools, Options dialog.

After you do so, every new Code window you open will have the keywords “Option Explicit” automatically entered in the General code segment. All variables in the form will need to be declared before you can use them.

To declare a variable, it would be helpful to know what data types Visual Basic has, wouldn't it? See **Table 30.1**.

**Table 30.1.** Variable types in Visual Basic 6.0.

| Boolean  | Logical                                                   |
|----------|-----------------------------------------------------------|
| Currency | Floating point number with four decimal places            |
| Date     | Date and/or Time                                          |
| Double   | Large floating point (decimal) number (in the gazillions) |
| Integer  | Small integer                                             |
| Long     | Large integer                                             |
| Single   | Small floating point (decimal) number (in the billions)   |
| String   | We know these as character strings                        |

To declare a variable, use the DIM keyword, like so:

```
Dim bIsAlive as Boolean
Dim cNetPay as Currency
Dim dBirth as Date
Dim fNationalDebt as Double
Dim iCounter as Integer
Dim lWorldCitizens as Long
Dim fTaxRate as Single
Dim sName as String
```

You can also use the PRIVATE and PUBLIC keywords to control the lifetime of the variable. A Private variable has scope throughout the routine, form, or module in which it was declared, while Public variables are available to any code anywhere in the project in which they are declared.

```
Public sName as String
Private sName as String
```

I'll beat on this for a bit, because strongly typed variables are a significant difference between VB and VFP. Once you have an Option Explicit declaration in your code (and you can type it in manually in the General code segment if you didn't check the Require Variable Declaration check box), you must declare every variable you use.

This prevents two undesirable behaviors from occurring. First, you can't accidentally create a bug by mistyping the name of a variable. In the following code, we are trying to assemble a message, sMessage, from consecutive values of the string snewValue. However, sMessage is never modified, because the second-to-last line assigns the concatenation of the existing string, sMessage, and the new value, snewValue, to a mistyped variable, aMessage.

```
sMessage = ""
For iCounter = 1 to 10
  <some code>
  snewValue = <some function>
  aMessage = sMessage + snewValue
End For
```

If you included an Option Explicit declaration in your code, you would have to declare sMessage, iCounter, and snewValue, and the compiler would detect an undeclared variable, aMessage. Bet you've done that once or twice in Fox, haven't you?

Second, by declaring the data type, you wouldn't accidentally change the data type of a variable. You've all done something like this:

```
dBirth = date()
<some code>
dBirth = 0
```

Strong typing of your variables will prevent this from happening again.

## Adding your own methods to forms and applications

Earlier I showed how you can add your own methods—functions or subprocedures—to a form by simply adding them in the General section of the Code window. Even if declared Private, those functions and subprocedures are now available throughout the form.

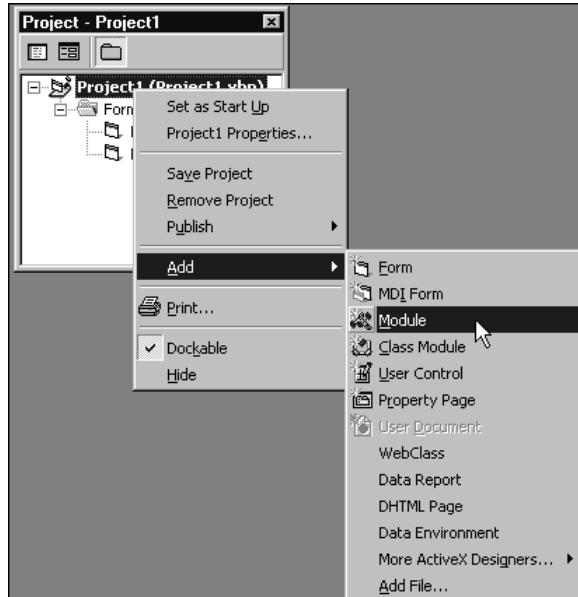
However, you will also want to create functions and subprocedures that are available globally throughout your application, much like common code in an .APP procedure file in Fox. In VB, these types of files are called modules. They are text files with .BAS extensions, and live in the Project Explorer alongside forms and other components. To create a module, open the Project Explorer (View, Project Explorer), right-click in the window, and select Add, Module from the context menus that display. See **Figure 30.11**.

You'll be prompted for the module to add—either an existing one or a new one, as shown in **Figure 30.12**.

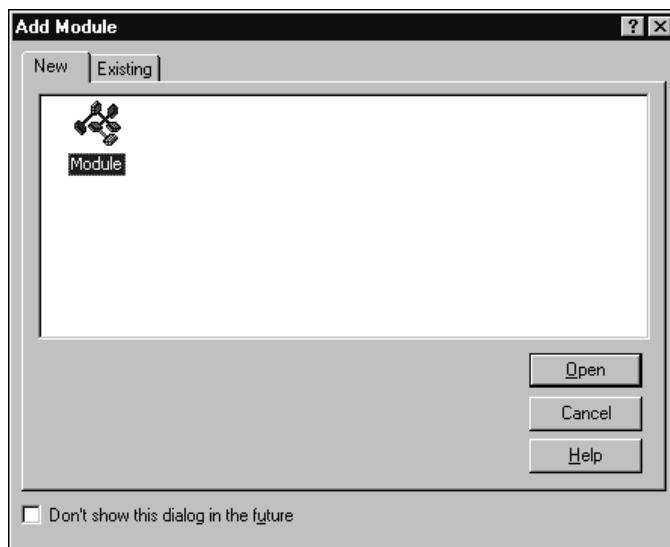
The module will be added to the Module node of the Project Explorer, as shown in **Figure 30.13**.

To create a subprocedure that's available in other parts of your project, simply create it like you did in a form:

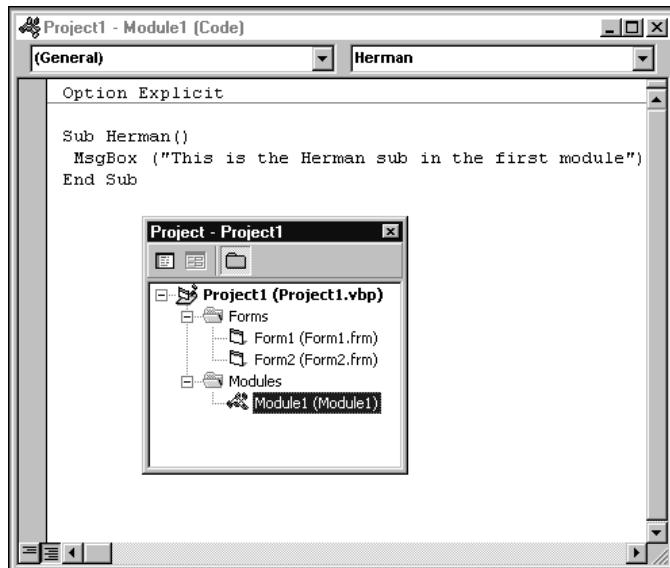
```
Sub Herman()
  MsgBox ("This is the Herman sub in the first module")
End Sub
```



**Figure 30.11.** Add a module to a project through the Project Explorer context menu.



**Figure 30.12.** You can choose to add a new module or an existing one to your project.



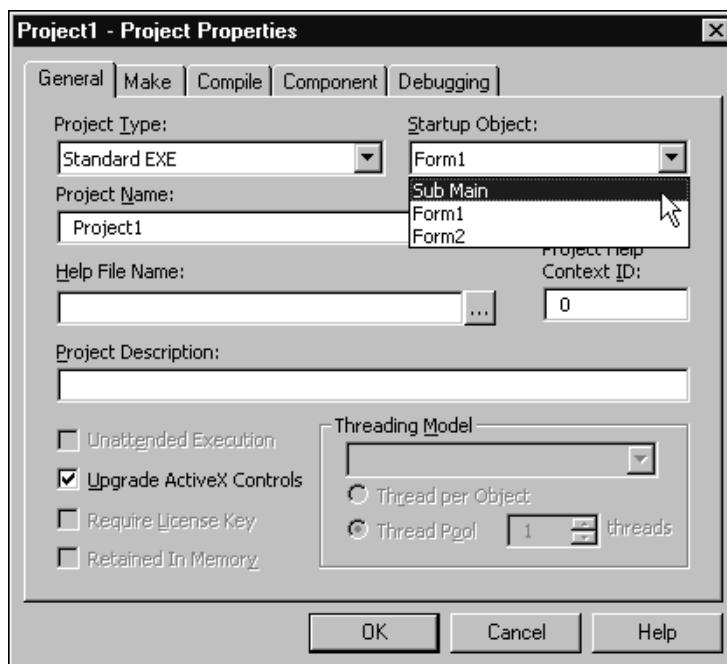
**Figure 30.13.** The Module node of the Project Explorer holds all modules for this project.

Then, to call the Herman subprocedure, use code like the following that's in the Click event of the Command2 command button on a form in the project:

```
Private Sub Command2_Click()
Call Herman
End Sub
```

Notice that the CALL keyword was required this time in order to run the subprocedure. This can vary, as you've seen earlier, depending on how you structure the syntax of the subprocedure and the way arguments, if any, are passed to it. Me? I like to use functions as much as possible, so that I can return a value that indicates whether the execution of a procedure was successful or not, so I don't worry about the CALL syntax at all.

You can also create a subprocedure in a module that serves as the startup program for your application. To do so, name it "Main." Then, if your project also has forms, you'll need to tell Visual Basic that it should run the Main subprocedure first, instead of the first form created in the project. To do so, right-click in the Project Explorer and select the ProjectN Properties menu command (where "ProjectN" is the name of your project.) Open the Startup Object dropdown in the General tab, and choose Sub Main as shown in **Figure 30.14**.



**Figure 30.14.** Set the Startup Object in the Project Properties dialog to the Sub Main subprocedure to make it run as the first component in your application.



**Stupid VB trick #4:** If you double-click a control to open the Code window for that control, the procedure for the Click event of that control will be displayed if there wasn't any code attached to that control to begin with. However, if you select that control, right-click to bring up the context menu, and select the View Code menu command, the same behavior does not occur.

## A command and function summary

So far, I've explored the Visual Basic IDE and explained how to create programs and forms. It's time to get to the fun part—learning the commands and functions that you can use to write programs and make forms actually do stuff.

What makes up a language? If you were to break down the basic elements that belong to every programming language, you'd end up with the following list: variables, operators, functions, expressions, and commands. I've already looked at a few examples of each of these; let's take a more in-depth look at them.

### The Immediate window

It's time to mention another component of the Visual Basic IDE that many VB programmers ignore, but that you'll use on a daily basis. What's one of the coolest things about Fox? The Command window, right? Type in a command, an expression, whatever—and you can see the results immediately. The VB equivalent is called the Immediate window. (Stop your snickering; I think the name is pretty cute, too.)

Open the Immediate window by selecting the View, Immediate menu command, or simply pressing Ctrl+G. We'll use this to work with functions and expressions.

### Variables

I've already gone into some detail about variables—you've learned about DIM and OPTION EXPLICIT and so on. You've also learned that there are a few data types in VB that are different than those in Fox. The trickiest part, to me, has been keeping the abbreviations for data types straight. See **Table 30.2**.

**Table 30.2.** Fox-to-VB data type abbreviations.

| Fox         | VB         |
|-------------|------------|
| c Character | s String   |
| l Logical   | b Boolean  |
| y Currency  | c Currency |
| d Date      | d Date     |
| t DateTime  |            |
| n Numeric   | f Double   |
|             | l Long     |
|             | f Single   |
| i Integer   | i Integer  |
| m Memo      |            |
| g General   |            |

Ever have one of those moments where you can't remember some obvious piece of data, like your mother-in-law's name, or your home phone number? The brain just does a total core dump all over the floor. With naming conventions between Fox and VB that seem to collide as often as they complement each other, I've given up trying to remember, and I just post a sticky note on my monitor.

Assigning values to variables is straightforward, with the possible exception of Booleans and Dates. I'll cover both of these in a couple of sections.

## Operators

Once you've got more than one variable, you're going to want to do something with both of them. For this, you need operators (or functions, which I'll hit next). The VB operators work just slightly differently than in Fox. Using the Immediate window, try entering the following:

```
fAmtBase = 100
fAmtAddOns = 43.2
? fAmtBase + fAmtAddOns
143.2
fAmtTotal = fAmtBase + fAmtAddOns
? fAmtTotal
143.2
fAmtDiscount = 12.4
? fAmtTotal - fAmtDiscount
130.8

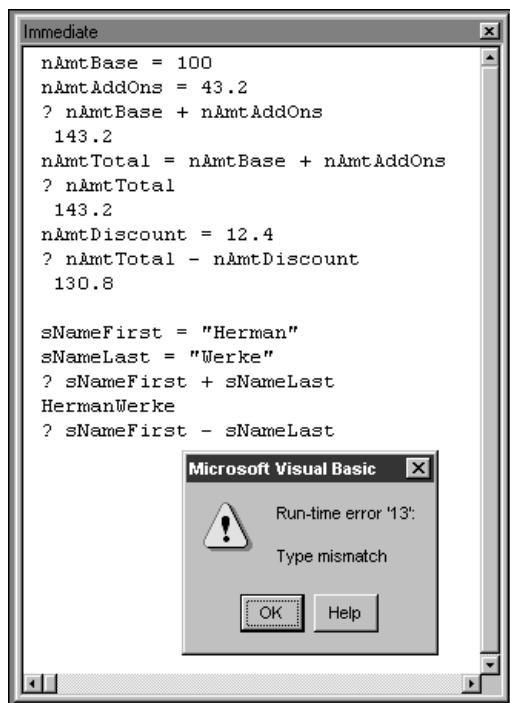
sNameFirst = "Herman"
sNameLast = "Werke"
? sNameFirst + sNameLast
HermanWerke
? sNameFirst - sNameLast
```

These operations all work as expected, until the last statement. In Fox, you can use the subtraction operator with two character strings, and not generate an error. Instead, you'll remove the trailing blanks from the first string, and then concatenate the two strings. In Visual Basic, however, doing so will generate an error, as shown in **Figure 30.15**.

The reason why is explained when you try the following lines of code:

```
s1 = "1111"
s2 = "2222"
? s1 + s2
11112222
? s1 - s2
-1111
```

As you see, Visual Basic will concatenate two text strings, but will then convert the strings to numbers, if possible, and then do arithmetic subtraction. If the strings can't be converted to numbers, you'll get an error message. To concatenate strings, then, you should use the “&” operator because, obviously, the + sign will mask mistakes if you don't have OPTION EXPLICIT declared.



**Figure 30.15.** Subtracting one string from another generates an error in Visual Basic.

Multiplication is just as straightforward, but division has a trick up its sleeve. A standard forward-leaning slash will divide two numbers; a backward-leaning slash will return the integral portion of the result. In this example, 1111/2222 results in a value of 0.5, and the integral portion is zero. Note that “\” doesn’t round—it truncates!

```

? s1*s2
2468642
? s1/s2
0.5
? s1\s2
0

```

Integer division runs considerably faster than decimal division, because there’s less work to do. No messing around with lots of places after the decimal point. If you’re doing division in a processor-intensive routine, such as in a loop, you should try to use integer division if possible.

Exponentiation is performed with a carat (^) just like in VFP.

Two other “special” characters, while not strictly operators, are the comment and code continuation characters. The “‘“ (single-quote) character denotes the beginning of a comment—probably easier to type than “\*\*” except that you’ve had years of practice with the asterisk

(unless, of course, you don't bother to use comments). The underscore denotes the continuation of a line of code, like so:

```
' This is a comment that precedes a two-line expression
? fAmtBase = (fAmtRawMaterial + fAmtLabor + fAmtBurden) _
  * (1.0 + fRate * fRateMultiplier)
```

### Booleans (logicals)

It seems that every language handles a logical True or False differently, and of course VB isn't any different. You simply assign "True" or "False" (without the quotes) to a Boolean variable, like so:

```
lIsAlive = True
lIsTalented = False
```

Note that these are not character strings!

### Dates

Dates are somewhat tricky until you grok the rules. First, the date delimiters are pound signs, not braces like in Fox. Second, VB assumes dates are entered as M/D/Y, with the current century. However, the display of dates is controlled by the Date tab in the Regional Settings applet in Control Panel. In other words, VB dates are always stored as m/d/yyyy but Windows controls how they are presented to the world.

```
d1 = #1/2/98#
? d1 + 5
1/7/98
' change the Short Date Style in the Date tab of Regional Settings to m/d/yyyy
d1 = #1/2/2098#
? d1 - 4
12/29/2097
```

### Times

Times work similarly to dates, in that the pound signs are the delimiters. For example, to assign the time "10 past noon" to a variable, you would use the following code:

```
dLunchTime = #12:10:17 PM#
```

Doing arithmetic with time is a little trickier. For example, to add 12 minutes and 13 seconds to a time variable, you'd use an expression like so:

```
? dLunchTime + #0:12:13#
12:22:30 PM
```

### Variants

Even if you use strongly typed variables throughout your Visual Basic career, you will need to have variables whose data type varies. For example, when you provide a text box on a form for a user to enter data into, you won't necessarily know what values are entered. You would store

the value from this text box to a special type of variable called a variant, and then proceed from there, depending on what type of data the user entered.

You will run into two prefixes for a variant—var and vnt. When you want to declare a variable as a variant, use the statement:

```
Dim varMySchizoVariable as variant
```

VB has functions that determine what's in a variant variable, and these will be covered shortly. By the way, given a choice, you should strive to stay away from variants—they are processed slower than other data types because VB has to figure out what is in the variable before it can do anything else.

## Constants

One practice to avoid in all programming practices is the use of “magic numbers.” These are where you run into a line of code like this:

```
fWeightShip = fWeightOriginal + 3117
```

Obviously, the adder 3117 means something, and most likely, when the line of code was written, the meaning of the number 3117 was obvious to everyone. Six months later, to a rookie programmer who just joined the company, the number 3117 looks like Bob Gibson's lifetime strikeout record and nothing more.

You can avoid this by defining a “constant” that is then used throughout the program or application, like so:

```
' at beginning of module, for example
Const CARDBOARD_CONTAINER_STANDARD_WEIGHT = 3117
' much later in the code
fWeightShip = fWeightOriginal + CARDBOARD_CONTAINER_STANDARD_WEIGHT
```

Then, when the weight of the cardboard container changes, you change it in only one place.

## Static variables

A static variable is one that can be accessed only by the procedure or module in which it was created, but that retains its values for the life of the application. This might feel a bit funny—it's like a hybrid of a global and a local at the same time. For example, suppose you had a routine that was called from various places in your application, but the variables in the routine had to have a “memory” from one call to another. Thus, when this routine was called from Module A, static variable sPreferenceValue was assigned the value of “ASCENDING.” This variable could be used throughout the routine, and, at some point, was changed to “DESCENDING.” As soon as the routine went out of scope, the variable sPreferenceValue was no longer visible anywhere in the application. Then, later on, Module B called this routine. At this time, sPreferenceValue still had the value “DESCENDING.”

So you can think of a static variable as being hidden, but not gone. You declare a static variable like so, using the “static” keyword instead of “dim” to define it:

```
Static iHowMany as Integer
```

## Arrays

An array is a collection of variables, similar to VFP, except that each array element must contain the same type of data. Static arrays have a fixed length, while the size of Dynamic arrays can be changed. To create a static array, use this statement:

```
Dim asMyArrayOfStrings(20) as String
```

Obviously, this array has 20 elements. Whoa! Not so obviously! It actually contains 21 elements because VB starts referencing array elements with 0, not 1, as in VFP. You’ll also notice I used parentheses, not square brackets, to surround the number of elements. Unfortunately for those of you who’ve practiced using brackets in array handling so as to keep arrays separate from function calls in VFP, you can’t use square brackets in VB. That’s the way it goes.

To define a Dynamic array, use the statement:

```
Dim asMyDynamicArrayOfStrings() as String
```

To define how many elements are in the array, use the ReDim statement:

```
ReDim asMyDynamicArrayOfStrings(13)
```

Here’s a gotcha for those of you who are used to VFP’s behavior of enlarging arrays, always keeping the existing contents: VB doesn’t do the same thing by default. You need to include the Preserve keyword so that you don’t blow away the contents of the existing array:

```
ReDim Preserve asMyDynamicArrayOfStrings(200)
```

You’ll note that I used a two-character prefix for the array—one identifying it as an array and the other identifying the data type in the array. The first isn’t technically necessary, since the parentheses ought to tell you you’re working with an array, but I like to be explicit—it’s just too easy to make a “dumb mistake” (as opposed to the intelligent kind of mistake). And the second one isn’t often used in VFP because an array can contain many different types of data—but in VB, you’re limited to a single type of data in an array (unless, of course, you define it as a variant array).

## Functions

Functions are, like in VFP, “mini-programs” that may or may not take input values and then return values. Just like with operators, keeping functions in VB straight from VFP functions takes a bit of concentration (or luck). There are well over 100 functions in VB 6—considerably fewer than in VFP, but still more than could be covered in this chapter. Let’s look at the different types of functions, and discuss some of the more useful ones.

### Data conversion

There are a number of functions that convert one type of data to another. The first, strictly, doesn't convert, but it helps you handle variants. VarType(varArgument) returns a value according to what you feed it, as shown in **Table 30.3**.

**Table 30.3.** Data type conversion functions in Visual Basic.

| VarType Value | Intrinsic Constant | Description                                        |
|---------------|--------------------|----------------------------------------------------|
| 0             | vbEmpty            | There is nothing in the variant.                   |
| 1             | vbNull             | There is not any valid data in the variant.        |
| 2             | vblnteger          | Contains a standard integer.                       |
| 3             | vbLong             | Contains a long integer.                           |
| 4             | vbSingle           | Contains a single precision floating point number. |
| 5             | vbDouble           | Contains a double precision floating point number. |
| 6             | vbCurrency         | Contains currency.                                 |
| 7             | vbDate             | Contains a date or time.                           |
| 8             | vbString           | Contains a string.                                 |
| 9             | vbObject           | Contains an object.                                |
| 10            | vbError            | Contains an error object.                          |
| 11            | vbBoolean          | Contains a Boolean value.                          |
| 12            | vbVariant          | Contains an array of variants.                     |
| 13            | vbDataObject       | Contains a data access object.                     |
| 14            | vbDecimal          | Contains a decimal value.                          |
| 17            | vbByte             | Contains a byte value.                             |
| 36            | vbUserDefinedType  | Contains a user-defined type.                      |
| 8192          | vbArray            | Contains an array of values.                       |

You can use the intrinsic constants in place of the numeric values to make it easier to read your code. The two following lines are identical:

```
If vartype(varMySchizoVariable) = 8 Then  
If vartype(varMySchizoVariable) = vbString Then
```

**Table 30.4** lists a number of conversion functions and what they do.

### Number handling

The Round() function works much like in VFP, but using a negative number as an argument in VFP will continue to round the number to the left of the decimal, while VB will generate an error.

### Data manipulation

The Choose() function has several similar functions in VFP; the first argument is used as an index to return a later argument in the list of parameters, like so:

```
Choose(index, parm1, parm2, parm3, parm4)
```

Unlike arrays, the index starts with the number 1.

**Table 30.4.** Visual Basic conversion functions.

| Conversion function | Description                                                                                                                                  |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| Cbool(argument)     | Converts an argument to a Boolean. Best used to convert a numeric value to True or False, such as non-zero to True and zero to False.        |
| Cbyte(argument)     | Converts an argument to a number between 0 and 255 if possible; otherwise converts to 0.                                                     |
| Ccur(argument)      | Converts an argument to a currency value if possible, using the Regional Settings to determine the number of decimals in the currency value. |
| Cdate(argument)     | Converts an argument to a date.                                                                                                              |
| Cdbl(argument)      | Converts an argument to a double precision number.                                                                                           |
| Cdec(argument)      | Converts an argument to a decimal.                                                                                                           |
| Cint(argument)      | Converts an argument to an integer (truncating, not rounding the decimal).                                                                   |
| Clong(argument)     | Converts an argument to a long integer (truncating, not rounding the decimal).                                                               |
| Csng(argument)      | Converts an argument to a single precision number.                                                                                           |
| Cstr(argument)      | Converts an argument to a string.                                                                                                            |
| Cvar(argument)      | Converts an argument to a variant.                                                                                                           |

The Asc() and Chr() functions return an integer representing the ASCII character code of the first character in the argument, and the character associated by the ASCII character, respectively, just like in VFP. Join() will concatenate the elements in an array, using a specified delimiter. Split() returns an array containing a number of sub strings.

The Array() function converts a comma delimited list of values to an array, like so:

```
Dim varMyString As Variant
varMyString = Array("String 1", "BBBBB", "3")
MsgBox (varMyString(0)) & Chr(13) _
& (varMyString(1)) & Chr(13) _
& (varMyString(2)) & Chr(13)
```

Note that the arguments inside the varMyString variable start with 0, not with 1! This will bite you sooner or later—you have been warned.

## Machine and environment

CurDir() returns the current directory. Dir() returns the name of a file, directory or folder that matches a specified attribute.

## Date and time

Date() returns the current system date. Now() returns the current date and time. DateAdd() returns a value to which a specific time interval has been added, while DateDiff() returns the difference between two dates. DatePart() returns a specific part of a date. Day(), Month(), Year(), Second(), Minute(), and Hour() all return components of a date argument. MonthName() and WeekdayName() do what they sound like—return a string indicating the month name or day name.

### **String manipulation**

InStr() returns the position of the first occurrence of one string in another. InStrRev() does the same thing, but starting from the end of the string. Left() returns the leftmost characters from a string; Right() does the same thing from the right. Ltrim(), Rtrim(), and Trim() all rid spaces—leading or trailing—from a string. Len() returns the number of characters in a string. Space() returns a string full of spaces. StrComp() compares two strings to return the result, depending on a specified type of comparison. String() repeats a given string N times. StrReverse() returns a string in reverse order. Format() (and a number of variations) returns a string containing a formatted expression. Replace() will substitute a string for another, a specified number of times. Ucase() will convert a string to uppercase; lcase() converts it to lowercase.

### **Math functions**

VB contains a standard set of math functions, including all of the geometry functions, Abs() (absolute value), Exp() (Exponentiation), Int() (Integer), Log(), Rnd() (random number), and Sqr() (square root).

### **Special functions**

MsgBox() creates a message box, similarly to MessageBox() in VFP. InputBox() is similar to MsgBox, except that it allows the user to enter a single value in a text box. Switch() evaluates a list of expressions and returns an expression associated with the first expression in the list that is True. Error() returns the message that corresponds to an error number. VB, being tied much closer to Windows, also has a full complement of functions that talk to various components of the Windows environment, but I won't go into those here.

## **Commands**

I know I claimed earlier that there were only 19 keywords in all of Visual Basic, but, well frankly, I was lying. There are actually over 70. And while that number pales to the 500+ commands and functions in VFP, it's still too many to simply list in an alphabetical manner and expect you to grok them in a single sitting. How to organize them?

I asked my family (a bunch of VB candidates if I ever saw them) for suggestions, and they came up with a number of interesting ones. One family member suggested listing them by length of word, another suggested listing them in the order in which they were introduced to VB, and a third suggestion was to sum up the ASCII codes that make up each word, and use the eventual values as a sort order. The fourth suggestion was "Winnie the Pooh" but the offerer of the suggestion is only 3 years old. Each of these suggestions was excellent (I have to say that or I'll end up sleeping on the couch until VB 43 is released), but it struck me that you use commands to do things when programming, and thus it makes sense to divide up the command into functional groups. So that's what I'll do here.

### **Logic structures**

You can do the same things with Visual Basic as you can with VFP in terms of controlling flow within a program module. The syntax is just a little different.

### ***Making a decision***

The statement for making a decision is, as it is with VFP, IF. However, VB's IF is much more flexible, as shown in this code fragment:

```
IF <condition1> THEN
<stuff to do>
ELSEIF <condition2> THEN
<stuff to do>
ELSEIF <condition3> THEN
<stuff to do>
ELSE
<stuff to do>
END IF
```

You'll notice that a "THEN" keyword is required after the IF <condition> expression, and the closing expression is two words, not just one as in VFP. The coolest thing about this is that you can nest multiple IF conditions using the ELSEIF <condition> expression instead of having to build multiple IF/ELSE/ENDIF constructs as you do in VFP. Also take note that each condition can be different!

### ***Choosing between multiple choices***

The SELECT CASE construct is VB's version of DO CASE in VFP. It, too, is a bit more powerful, as shown in the following code fragment:

```
SELECT CASE sFileExtension
  <stuff to do>
CASE "BAT"
  <stuff to do>
CASE "SYS"
  <stuff to do>
CASE "DBF", "CDX", "FPT"
  <stuff to do>
CASE "MAA" to "MZZ"
  <stuff to do>
CASE ELSE
  <stuff to do>
END SELECT
```

This construct takes as input a string expression, sFileExtension, that contains the extension of a file, and processes it according to what extension it is. Thus, .BAT files get one treatment, .SYS files get a different treatment, and so on. This is different from VFP in that you can use multiple, inconsistent conditional expressions following each CASE statement in VFP, and, thus, you have to use the entire conditional expression. Note that you can include multiple values following VB's CASE statement, as in the third CASE statement. The code following that piece will be executed for .DBF, .CDX, and .FPT files. In VFP, you'd have to write an expression like so:

```
CASE inlist(cFileExtension, "DBF", "CDX", "FPT")
```

Note that you can also have VB span a series of values, as in the fourth CASE where every extension between MAA and MZZ will be treated the same. CASE ELSE works the same as OTEHRWISE in VFP.

### **Looping**

Somebody in Redmond went bonkers when it came time to assemble looping constructs in VB, because there are five of them. I'll describe the first four and then explain why you shouldn't use the fifth.

The FOR NEXT construct is the basic looping construct, as shown in this code fragment:

```
FOR nIndex = 1 to 10 STEP 2
  <some code to run>
  IF <some bad condition occurred> THEN
    EXIT FOR
  END IF
  <more code could be here>
NEXT
```

You'll see that NEXT is used in place of END FOR in VFP, although many of you are probably using the (sort of new) NEXT keyword in VFP instead of END FOR. (If you're not, you should, because it will cut down on the confusion generated when switching between languages.) You'll also see that the escape route out of a FOR loop is EXIT. Big surprise.

If your index expression is an element in a collection of some sort, you can use FOR EACH much easier:

```
FOR EACH <element> IN <group>
  <some code to run>
  IF <some condition found the right element> THEN
    EXIT FOR
  END IF
  <more code could be here>
NEXT
```

So far, so good. The next two logic structures are pretty similar: DO WHILE and DO UNTIL. In fact, you can probably noodle them out yourself, right? Not so fast!

```
DO WHILE <condition>
  <some code to run>
LOOP

DO UNTIL <condition>
  <some code to run>
LOOP
```

The trick to both of these is that, while the above syntax looks comfortable to you, you're most likely not going to see it in experienced VB-ers' code. That's because you can also put the WHILE and UNTIL expressions after the LOOP keyword, like so:

```
DO
  <some code to run>
LOOP WHILE <condition>

DO
  <some code to run>
LOOP UNTIL <condition>
```

Obviously, doing so changes how the loop is processed—much like incrementing a counter at the beginning or the end of a loop. You can use the EXIT DO statement to get out of DO JAIL early, like so:

```
DO WHILE <condition>
  <some code to run>
  IF <some bad condition occurred> THEN
    EXIT DO
  END IF
LOOP
```

The WHILE/WEND construct is a cheap, poorly imitated version of DO WHILE, because you can't use EXIT DO to escape early. If that's not important to you, then you can use WHILE/WEND instead. Keep in mind, however, that if your requirements change and you have to be able to terminate processing early, you'll have to recode your WHILE/WEND to DO WHILEs. Why not save yourself the inevitable hassle now?

Instead of trying to describe each VB statement in detail, I've just listed the rest of them, so that you can get a quick idea of what functionality is available to you natively, and, thus, what you'll have to write yourself or live without.

### **File handling**

|              |                                                                                                                                               |
|--------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| Close        | Concludes IO to a file opened using OPEN.                                                                                                     |
| Get          | Reads data from an open disk file into a variable.                                                                                            |
| Input #      | Reads data from an open sequential file and assigns the data to variables.                                                                    |
| Let          | Assigns the value of an expression to a variable or property.                                                                                 |
| Line Input # | Reads a single line from an open sequential file and assigns it to a string variable.                                                         |
| Lock, Unlock | Controls access by other processes to all or part of a file opened using OPEN.                                                                |
| Lset         | Left-aligns a string with a string variable, copies a variable of one user-defined type to another variable of a different user-defined type. |
| Mid          | Replaces a specified number of characters in a variant (string) variable with characters from another string.                                 |
| Open         | Enables IO to a file.                                                                                                                         |
| Print #      | Writes display-formatted data to a sequential file.                                                                                           |
| Put          | Writes data from a variable to a disk file.                                                                                                   |
| Reset        | Closes all disk files opened using OPEN.                                                                                                      |
| Rset         | Right-aligns a string within a string variable.                                                                                               |
| Seek         | Sets the position for the next read/write operation within a file opened using OPEN.                                                          |
| Width #      | Assigns an output line width to a file opened using OPEN.                                                                                     |
| Write #      | Writes data to a sequential file.                                                                                                             |

**Program control**

|                   |                                                                                    |
|-------------------|------------------------------------------------------------------------------------|
| Call              | Transfers control to a Sub procedure, Function procedure or a .DLL.                |
| Function          | Declares the name, arguments, and code that form the body of a function procedure. |
| GoSub             | Branches to and returns from a subroutine within a single procedure.               |
| GoTo              | Branches unconditionally to a line within a procedure.                             |
| On GoSub, On GoTo | Branches to a line depending on the value of an expression.                        |
| Stop              | Suspends execution of a program.                                                   |
| Sub               | Declares the name, arguments, and code that form the body of a sub procedure.      |
| With              | Executes a series of statements on a single object or user-defined type.           |

**File system**

|             |                                                                            |
|-------------|----------------------------------------------------------------------------|
| ChDir       | Changes the current directory or folder.                                   |
| ChDrive     | Changes the current drive.                                                 |
| FileCopy    | Copies a file.                                                             |
| Kill        | Deletes files from a disk.                                                 |
| MkDir       | Creates a new directory.                                                   |
| Name        | Renames a directory or file.                                               |
| RmDir       | Deletes a directory.                                                       |
| SavePicture | Saves a graphic from the picture or image property of a control to a file. |
| SetAttr     | Sets attributes information for a file.                                    |

**Variables**

|                 |                                                                                                                                                      |
|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| Const           | Declares constants for use in place of literal values.                                                                                               |
| Declare         | Declares references to external procedures in a .DLL—used at module level.                                                                           |
| Deftype         | Sets default data type for variables, arguments passed to procedures, and return type for Function and Property Get procedures—used at module level. |
| Dim             | Declares variables and allocates storage space.                                                                                                      |
| Enum            | Declares a type for an enumeration.                                                                                                                  |
| Erase           | Reinitializes the elements of a fixed-size array and releases dynamic array storage space.                                                           |
| Option Base     | Declares the default lower bound for array subscripts—used at module level.                                                                          |
| Option Compare  | Declares the default comparison method to use when string data is compared—used at module level.                                                     |
| Option Explicit | Forces explicit declaration of all variables in that module—used at module level.                                                                    |
| Option Private  | Prevents a module's contents from being referenced outside its project.                                                                              |
| Private         | Declares private variables and allocates storage space—used at module level.                                                                         |
| Property Get    | Declares the name, argument, and code that form the body of a property procedure that gets the value of a property.                                  |
| Property Let    | Declares the name, argument, and code that form the body of a property procedure that assigns the value to a property.                               |
| Property Set    | Declares the name, argument, and code that form the body of a property procedure that sets a reference to an object.                                 |
| Public          | Declares public variables and allocates storage space—used at module level.                                                                          |
| Randomize       | Initializes the random-number generator.                                                                                                             |
| ReDim           | Reallocates storage space for dynamic array variables.                                                                                               |
| Static          | Declares variables and allocates storage space—used at the procedure level.                                                                          |
| Type            | Defines a user-defined data type containing one or more elements—used at module level.                                                               |

**Registry**

|               |                                                                                   |
|---------------|-----------------------------------------------------------------------------------|
| DeleteSetting | Deletes a section or key setting from an application's entry in the Registry.     |
| SaveSetting   | Saves or creates an application entry in the application's entry in the Registry. |

**Classes**

|            |                                                                                                   |
|------------|---------------------------------------------------------------------------------------------------|
| Implements | Specifies an interface or class that will be implemented in the class module in which it appears. |
| Load       | Loads a form or control into memory.                                                              |
| Set        | Assigns an object reference to a variable or property.                                            |
| Unload     | Unloads a form or control from memory.                                                            |

**N.E.C. (Not Elsewhere Classified)**

|            |                                                                                                 |
|------------|-------------------------------------------------------------------------------------------------|
| Date       | Sets the system date.                                                                           |
| Error      | Simulates an error.                                                                             |
| Event      | Declares a user-defined event.                                                                  |
| On Error   | Enables an error-handling routine and specifies the location of the routine within a procedure. |
| Resume     | Resumes execution after an error-handling routine is finished.                                  |
| Rem        | Identifies a comment.                                                                           |
| RaiseEvent | Fires an event.                                                                                 |
| SendKeys   | Sends one or more keystrokes to the active window as if typed at the keyboard.                  |
| Time       | Sets the system time.                                                                           |

If you look through these commands, you'll see that they seem to stop short of some of the functionality you're used to in VFP. That's because much of the language is a throwback to BASIC before it became "Visual Basic," and thus still contains legacy references to opening and closing files and such. Most of the work you'll do in VB now will be done visually—connecting to data will be done by dropping data-aware controls on a form, not through a series of commands like "USE" and "APPEND FROM" like VFP still does.

And one last thing: Remember that you can't abbreviate to the first four characters as you can in VFP!



**Stupid VB (Windows) trick #5:** *The DATE statement sets the current system date. That seems reasonable. However, it appears that the emphasis that Microsoft puts on the term "cross platform" is well warranted, because there is a difference in permissible date specifications depending on whether the operating system is Windows 95/98 or Windows NT. Windows 95/98 can handle system dates between 1/1/1980 and 12/31/2099. However, Windows NT can only handle dates between 1/1/1980 and 12/31/2079. Hmm, obviously a conspiracy. Do you think maybe this means that Microsoft is going to kill NT, and they're just stringing us along for 80 years? Hmm...?*

**Creating the user interface: VB forms and controls**

Now that we have covered the language, we've got a good foundation under our belt. It's time to look at the tools that we'll use to create the most interesting part of the application—from the user's perspective, that is. Much like an automobile and an 18-wheeler, some of the tools and

techniques will be familiar to experienced VFP developers, but other techniques and components are either brand new, or just simply different. Let's explore.

It's difficult to do justice to the topic of "user interface" in a few pages, so I'm going to blast through this, and cover the big differences in two areas. First, I'll discuss the mechanisms you use to manipulate forms and controls in Visual Basic, and then I'll go through the more common controls and compare and contrast them with their VFP brethren.

### **Creating a form and dropping a control on it**

Silly enough, it took me the better part of a day to get the hang of putting controls on a VB form. You don't click on the control in the toolbox and then on the form, and expect the control to be drawn for you like in VFP. Instead, either click on the control, and then click and drag the control on the form—just clicking on the form doesn't do anything. Alternatively, you can just double-click on the control in the toolbox—the control will be placed in the center of the form. If you double-click on a control in the toolbox multiple times, you'll end up with several controls on top of each other in the center of your form.

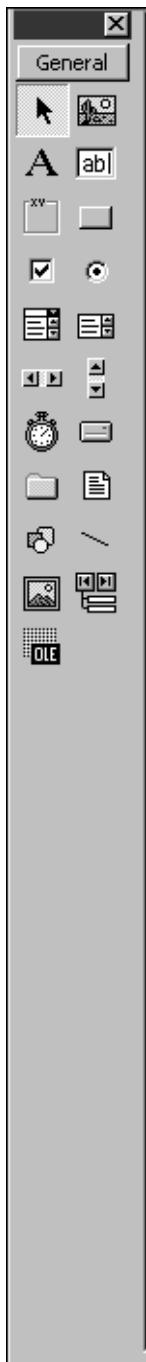
Moving a control is a little different—the cursor keys don't nudge a selected control or group of controls bit by bit. Instead you have to hold down the Ctrl key to nudge them one way or another. You'll also want to take note of the Format, Lock Controls menu option—selecting it will turn all of the sizing handles to white, and you'll not be allowed to use the mouse to move or resize the controls from then on. Evidently accidental moves by novice VBers was a problem at some point in time. You can, however, still use the keyboard to move and resize as you want.

Alignment is one of those "half-full, half-empty" issues. You can use the Tools, Options command to display or hide the grid on a form, define the granularity of grid lines, and to have controls snapped to grid lines or not. You can also use the Format, Align menu command to align a group of selected controls, much as you do in VFP. And if you're like me, you never use the menu command in VFP, preferring to use the Layout toolbar. Unfortunately, if you look for a Layout or an Alignment toolbar in VB, you won't find one. Instead, select the Form Editor toolbar. It looks sparse, but you'll see that each button opens up into a bunch of related choices, which I think is pretty cool. It just took a while to get used to it—it's still two mouse clicks instead of one.

### **VB's intrinsic controls**

Visual Basic is considerably different from VFP in that hardly any VB developers just accept the controls that come "in the box" as all they'll use. The whole idea behind VB was to allow the developer to extend the environment by using additional controls as they desired—truly, very few VB developers' toolboxes look alike.

However, there are 20 controls that come with VB and appear automatically on the toolbox's toolbar. I'll talk about most of them here. If you dock the toolbox to the left of the VB IDE, you'll see them as shown in **Figure 30.16**.



The Label control is similar to VFP's label, and has many of the same properties.

The TextBox control is similar to its VFP counterpart, but also does double duty to serve as VFP's EditBox match. The Value property in VFP is called the Text property in VB for this control. You can set the MultiLine property to True and then enter more than one line of text—where you'd use an EditBox in VFP to do the same thing. If you have MultiLine set to True, you can use the ScrollBars property to display them as desired.

The CheckBox control is similar to VFP's CheckBox. It can contain a value of 0 (not checked), 1 (checked), or 2 (disabled). You have to be careful about using a value of 2, because once a check box's value has been set to 2, you can't tell whether it's checked or unchecked. 2 (Disabled) is also different from setting the Enabled property to False; the former keeps the check box's caption enabled (black) while the latter also dims the caption, turning to a fuzzy gray/white combination.

The CommandButton is VB's answer to VFP's command button—and a button is a button.

The OptionButton sort of maps to an option button in VFP, but there are a couple of significant differences. If you've already placed an option button on a VB form, you might be wondering, "Where's the Option Group control?" And the answer is ... there isn't any! All of the option buttons you place on a form—regardless of where on the form—are part of the same option group. Blech, you say? Me, too. It is a bit easier, of course, to be creative with placement—don't you hate having to enlarge an option group, and then edit the group, and then align the buttons where you want them? Kind of a pain. In VB, this is easier.

But the flip side is that if you want more than one option group on a form, you have to build your own container first, using a Frame control, and then add individual option buttons. Like I said, "Blech."

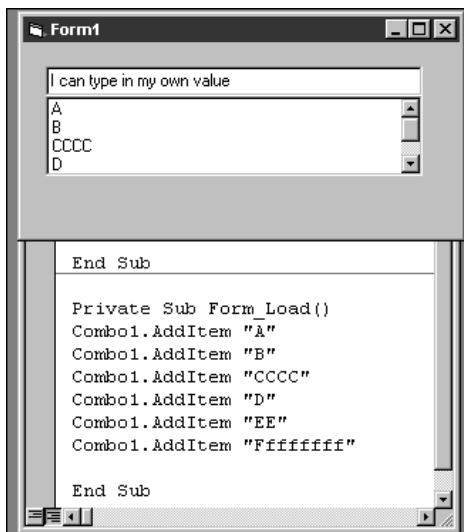
A group of option buttons (I didn't say "an Option Button Group," did I?) works just like you would expect—selecting one deselects another that had been selected. You need to write code to deselect all of them, like the following fragment (you could put this in the Click() method of a command button, for example):

```
Option1.Value = 0  
Option2.Value = 0  
Option3.Value = 0
```

The ComboBox and ListBox controls are similar to VFP's Combo Box and List Box controls, and I think they're actually somewhat easier to use "out of the box."

**Figure 30.16.** Visual Basic comes with 20 native controls.

The ComboBox control is similar to VFP's combo box, complete with multiple modes and a variety of properties. You can set the Style property to one of three possibilities. 0 (DropDown Combo) means you can enter text or click on the arrow to open the combo. 1 (Simple Combo) means the combo box is always open but you can also enter text if you want. It's a little unusual—you need to resize the combo box to display more than one item, and the result ends up looking like a text box with a list box placed below it on the form. See **Figure 30.17.** 2 (DropDown List) means that you can't enter text, and you have to open the control in order to select a value.



**Figure 30.17.** A Simple combo box looks like a text box and a list box together.

The easiest way to add items to a ComboBox or ListBox is with the AddItem method:

```
List1.AddItem "A"
List1.AddItem "B"
List1.AddItem "cccccc"
```

You can make a ListBox multi-selectable by setting the Mult-Select property to 1 (Simple) or 2 (Extended). Simple means that you can just keep clicking on items and they stay selected (clicking a second time deselects the item). Extended means you can use the Ctrl and Shift keys to select a range of items—such as the 3rd, 4th, 5th, 9th, and 12th.

You can determine which item has been selected in a ListBox with the Text property (just as you would use the DisplayName property in VFP). If you have Multi-Select set to 1 or 2, the Text property displays the last item selected. In order to determine all those items selected, use the Selected property together with the ListCount property, much as you would in VFP:

```
Dim nItemNumber As Integer
nItemNumber = 0
Do While nItemNumber < List1.ListCount
    If List1.Selected(nItemNumber) = True Then
        MsgBox "Hey, I'm selected " & nItemNumber + 1
    End If
    nItemNumber = nItemNumber + 1
Loop
```

Note that the index of the array populating the ListBox starts with zero, not 1 as in VFP.

The Shape and Line controls allow you to create a variety of lines, circles, and rectangles on your forms, much like the same controls in VFP do. The Shape property of the Shape control allows you to choose between Rectangle, Square, Oval, Circle, Rounded Rectangle, and Rounded Square.

The Image and Picture controls allow you to place graphic files on your form. The Image control is lightweight, but doesn't allow you to overlap controls (and thus, images), and can't receive focus. The Picture control, on the other hand, can be overlapped and can receive focus—which makes it a good candidate to use when creating graphical controls.

The DirListBox control, in combination with the DriveListBox and the FileListBox controls, allow you to build file navigation dialogs quickly and easily.

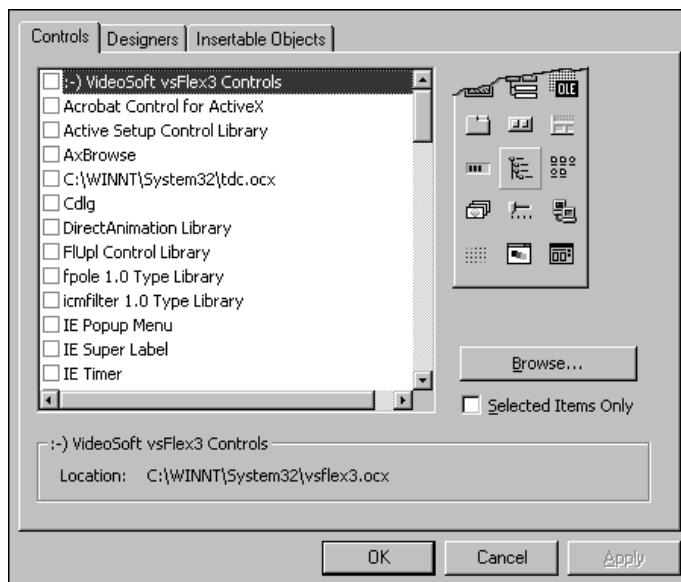
## More controls

There are, of course, more controls. Some ship with Visual Basic; others are produced by third parties and you have to beg, borrow, steal (or pay for!) them.

Those that ship with VB are also installed when you install VB; all you have to do is tell VB that you want access to them. To do so, right-click in the Toolbox and select the Components menu command. This opens the Components dialog as shown in **Figure 30.18**.

As with VFP, check off the controls you want to have displayed in the Toolbox, and then click OK. Note that this dialog is much nicer than the one we get in the Controls tab of the Tools, Options dialog in VFP. How much nicer? Oh, let me count the ways. First, you can see more of the name of the control than in VFP—and if the name is too long to fit in the list box, you can scroll the list box to the right and left. In VFP, you're just hosed. Second, you can see exactly which file on disk represents the control—in VFP, you're just supposed to guess, I guess. You can also choose between Controls, Designers and Insertable Objects in three different tabs. Finally, you can choose to see just those objects that you've selected—quite nice when you've selected three of 200 available. In VFP, get your pen and a piece of paper unless you've got a really good memory. (I've asked for these improvements, but you know the refrain—so many ERs, so little time.)

Once you've selected some controls, they'll show up in the Toolbox along with the original 20. You might not want your Toolbox littered with so many; right-click on the Toolbox, select the Add menu command, and add a tab to the Toolbox. Then drag your controls to the new tab. You can organize your controls any which way you want using this mechanism.



**Figure 30.18.** The Components dialog is used to add controls to the Toolbox.

## Incorporating data access into your VB app

What good is a programming language if you can't get to data somehow? Here is probably the biggest difference between Visual Basic and Visual FoxPro—VB doesn't have a native data engine. Thus, the way you talk to a data set is significantly different. However, you'll find that once you've made the connection to a data set, what you do next is more familiar than you would have thought. In this section I'll discuss the fundamental concept of how you go about accessing data in VB, and describe a couple of ways of making it happen. For those of you interested in the buzzwords of the month, I'll discuss the various TLAs involved in data access: ADO, DAO, RDS, RDO, UDS, ABC, XYZ and do-re-mi; and why you want to use ADO. Then I'll show you how to get started with ADO quickly and easily.

Visual Basic is a general-purpose programming language that has more and more been used as a front end to database applications. As such, it does not have a native data manipulation language—instead, data access is a feature that's been cobbled onto it, somewhat as an afterthought. Before you disparage this approach, you should know a little about how this state of affairs came about.

### A brief history of data access in Visual Basic

If you look back a dozen years or so, you had three general choices when writing a database application. You could use one of those "Xbase" tools like dBASE, Clipper, or Fox. You could use a non-Xbase tool like Paradox or Rbase. Or you could use BASIC and manipulate text or binary files by hand. Many programmers chose the third route—not only hand-coding screens

like Xbasers did in the early days, but also hand-coding data access. Instead of the Xbase “USE” command to open and gain access to a file, you had to go through a series of commands to identify the file, grab a file handle, open the file, write bytes to the file, reposition the pointer in the file, and so on—much like you can do with low-level file functions in Fox. Imagine if you were limited to LLFFs in Fox for all your data access!

When Visual Basic first arrived in the early ‘90s, hand-coding of screens went away—you simply drew objects with a Form Designer tool. Each subsequent version of VB added more power and capability, so no one noticed that they were still hard-coding the data access portion of the application.

In the mid-‘90s, more powerful data access mechanisms were added to Visual Basic, to hide the hand-coding of data access much like the Form Designer hid the hand-coding of screens. These mechanisms have gone through a number of iterations, and Microsoft is not done yet.

## **DAO, ADO, 123, Hike!**

It’s not worth spending a lot of time going back to the beginning of data access in VB, but the last two mechanisms are worthy of discussion. Many VB applications in use today were written using a mechanism called Data Access Objects, or DAO. DAO provided access to relational databases like Microsoft Access and SQL Server. It’s been superseded by ActiveX Data Objects, or ADO. ADO is actually a front end for a broader technology called OLE DB.

OLE DB has come about due to the explosion of non-relational data sources, such as e-mail, images, movies, sound files, HTML documents, and so on. Just as ODBC is a generic API for all sorts of relational databases, OLE DB is a generic API for all sorts of data. However, as has been said over and over again, writing to the OLE DB API is “just too hard” so ADO was put together to provide an easy-to-use interface to OLE DB.

While you can still use DAO, ADO is the current technology of choice, and unless you are faced with having to support old VB apps, I’d suggest you ignore DAO and learn ADO. That’s what I’m going to do here.

## **What is ADO?**

Just in case you skipped Chapter 28, I’ll repeat myself briefly here. ADO is just a .DLL that sits in your WINDOWS\SYSTEM or WINNT\SYSTEM32 directory. It provides services just like any other .DLL provides functionality. You can drop an ADO control on a form, just like you can drop a Calendar .OCX or a .DLL that provides serial communications. No real magic there—but it’s not terribly clear to those just getting started, or to those of us who are used to having data access built into the product like we are with Fox.

So, like JET, the data access engine used in Microsoft Access, ADO is just an engine—but it provides the ability to talk to a variety of data sources, as opposed to just an .MDB file like Access. This data engine, fortunately, was not written by the same people who wrote JET—instead, it was written by the folks who built and enhanced the Fox data engine. Yes, an ADO recordset is, in many ways, an abstraction of a Visual FoxPro cursor!

## Connections

When you issue a USE command in Fox, an awful lot happens behind the scenes, but eventually you're attached to a .DBF file. Implicit in this scenario is that the file you're "USEing" is a .DBF with a specific file structure. Other than that, a .DBF is essentially a dumb file just sitting on disk. If you're using a .DBF that's part of a .DBC, there's a bit of additional complexity in that the .DBC inserts a layer of information in between you and your data, but the Fox engine handles all that transparently. Thus, you can USE any kind of database as long as it's got a .DBF file structure.

With ADO, you can connect to a variety of data sources that can be structured a whole bunch of different ways, and in fact, can be housed in an entirely different environment, such as a SQL Server database or an e-mail program. Not only that, but it is theoretically possible to write a data access layer in an application, and then attach to it from a different front end. For example, as the story goes, you could write a middle tier in Visual FoxPro using ADO to talk to any old back end, and then use a front end written in, oh, say, HTML or Visual Basic. I don't know of anyone who has done this successfully (in other words, had a legitimate business need for this architecture, successfully implemented it with a bare minimum of compromises, and finally, actually made money at it), but it's an interesting intellectual exercise, and somewhere down the road I imagine it will become useful.

First, I'm going to create a simple data-entry and navigation form using the ADO Data Control, and in doing so, create a connection so that you can see where it fits in with the larger scheme of things. The ADO Data Control acts much like a wizard—set a few properties and you're done. However, just like a wizard, it's easy to use it without understanding what's underneath, and thus your usage is superficial and limited. For this reason, I'll then take apart the connection, and build one programmatically so that you can see how each piece works.

Just like every introductory demonstration, I'll use Microsoft Access 2000 as the back end. Oh, get that gloomy look off your face. First of all, it's more likely that you've got a copy of Access lying around than SQL Server, and second, there's less overhead and infrastructure to worry about. Once you've got the basics down, you can upgrade your choice of back end and deal with the added complexities then.

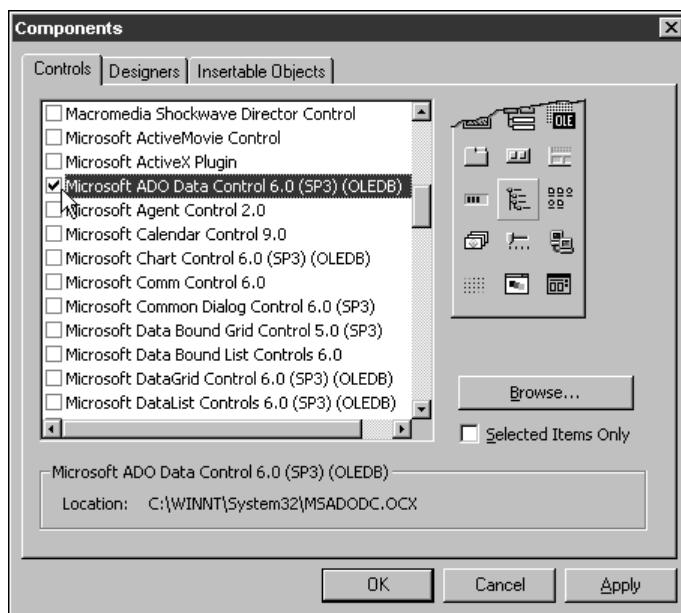
## Setting up a form with an ADO Data Control



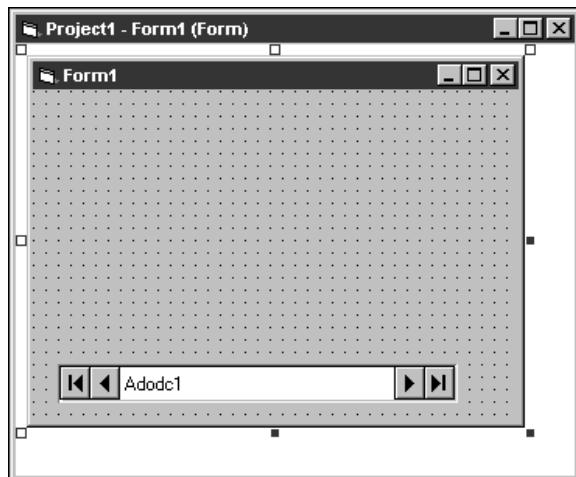
This example uses the MG.MDB database provided in the downloads for this chapter. It contains about six tables having to do with an automobile car club, including *tblCars*, which I'll use in this first example.

First, after opening Visual Basic, you'll need to add the ADO Data Control to your Toolbox. This control comes with Visual Basic, but is not part of the intrinsic Toolbox. Right-click on the Toolbox, select the Components menu option, and select the Microsoft ADO Data Control 6.0 (OLE DB) item in the list box in the Controls tab as shown in **Figure 30.19**. After you check the check box and select OK, you'll have another control on your Toolbox.

Next, open a new project (Standard EXE) and a blank form. Drop the ADO Data Control on the form. You'll get a navigation bar as the visible part of the control, as shown in **Figure 30.20**.



**Figure 30.19.** Add the Microsoft ADO Data Control 6.0 to your Toolbox before starting out.



**Figure 30.20.** Dropping the ADO Data Control 6.0 on your form will display a navigation bar.

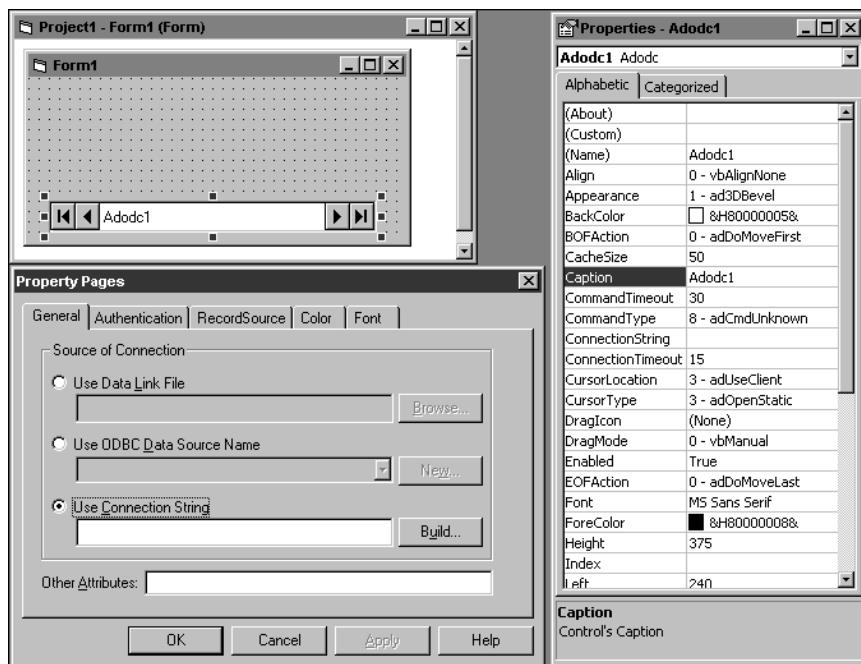
Next, it's time to connect this control to a data source. First, right-click on the ADO Data Control and select the ADODC Properties menu command. The Property Pages dialog appears. Note that this dialog is not the same as the Properties window that is also displayed, as shown in **Figure 30.21**. (You can also open the Property Pages dialog by double-clicking on the Custom property in the Properties window, or clicking the ellipsis button to the right of the Custom property.)

The Property Pages dialog acts like a “mini-wizard” where you are guided through each piece of building the connection. (I’ll explain what all of these options are for later.) Select the Use Connection String option button, and then click the Build button. The Data Link Properties dialog opens, and a list of OLE DB providers is displayed (see **Figure 30.22**).

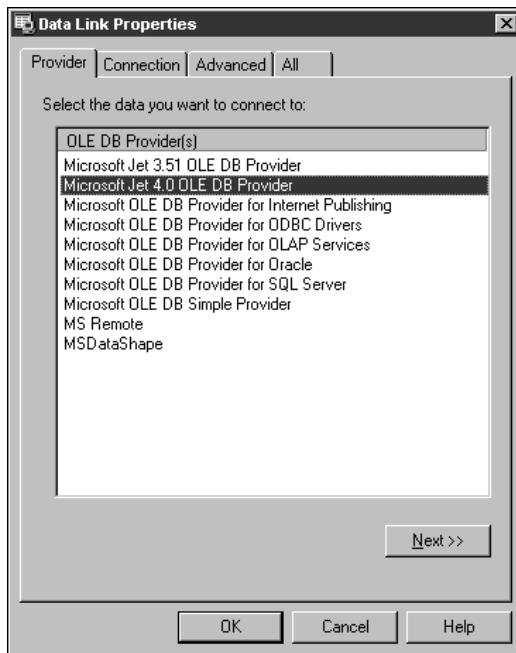
Remember that OLE DB is an API to a variety of data sources, just like ODBC is. The only difference is that ODBC basically attaches to relationally formatted data, while OLE DB is more robust. Pick the OLE DB provider you want—in this case, it will be Jet 4.0.

You might be wondering where these OLE DB providers come from. They just sort of come along for the ride. Some are installed with Windows, and others with a specific application. For example, when you install Office 2000 Pro (or, at least, Access 2000), you get the Jet 4.0 OLE DB Provider automatically.

Click the Next button to identify which Jet 4.0 database you want to work with.



**Figure 30.21.** Open the Property Pages dialog by right-clicking on the ADO Data Control and selecting ADODC Properties.



**Figure 30.22.** The Data Link Properties dialog displays all available OLE DB providers.

The contents of the Connection tab of the Data Link Properties dialog will vary according to which kind of OLE DB provider you selected in the previous tab. Note that in **Figure 30.23**, I've already selected the MG.MDB Access 2000 database that's residing somewhere deep in the bowels of drive G. Also note that the OLE DB Provider for Jet 4.0 knew to ask for a user name, a password, and to allow a couple of password-related options.

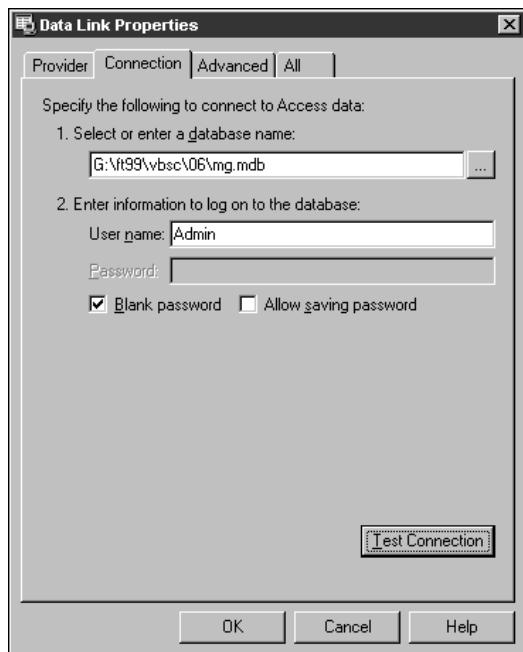
Once you've entered or selected a database, you might want to make sure that ADO can talk to it by clicking the Test Connection button.

Finally, click OK, and you're all set—you'll be returned to the Property Pages dialog, and you'll see a string of text entered into the text box below the option button. The whole string will read something like this (all on one line in the text box, of course):

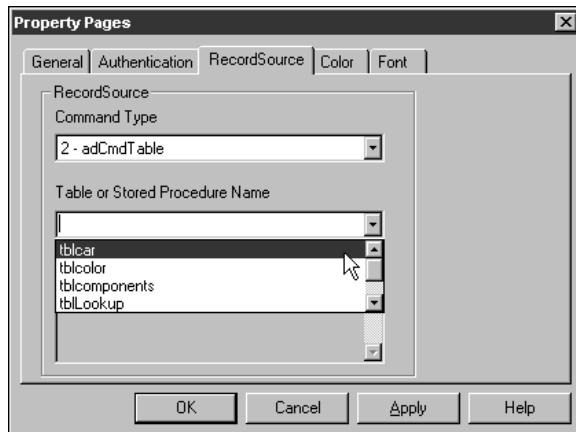
```
Provider=Microsoft.Jet.OLEDB.4.0;
Data Source=G:\ft99\vbsc\06\mg.mdb;
Persist Security Info=False
```

You'll see that the various items you selected are identified in the connection string, including a hard-coded data source. That's why I didn't provide a sample form with this application—you most likely wouldn't have a drive G with the same directories I have.

Finally, you'll need to identify a RecordSource. Select the third tab in the Property Pages dialog, change the Command Type to adCmdTable, and when you open the Table or Stored Procedure Name combo, you'll see all of the tables in the MG.MDB database displayed. Pick the table of interest—in **Figure 30.24**, I've picked *tblCar*.



**Figure 30.23.** Use the Connection tab to select the specific data source once you've selected an OLE DB provider.



**Figure 30.24.** Use the RecordSource tab to select the actual data in the database you picked in the previous step.

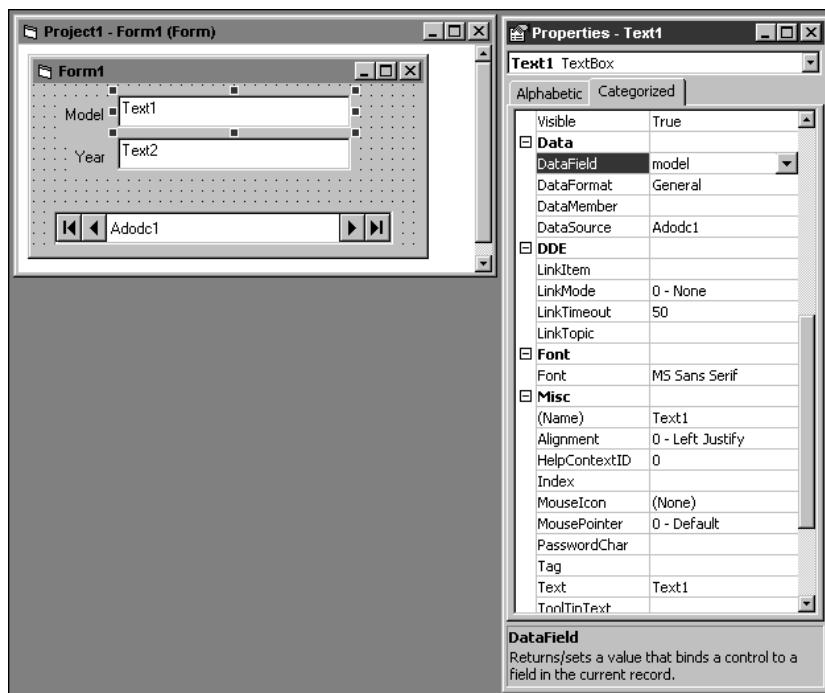
That's it for setting up the connection. Now it's time to put something more interesting on the form than just a data control. How about a text box or two?

## Connecting controls to an ADO recordset

Drop a couple of labels and text boxes on your form. In order to bind a text box to a field in a record source, you need to specify two properties. The first is to identify which ADO recordset contains the field you want. This might seem a bit odd at first, until you realize that an ADO recordset is much like a View in a VFP data environment. You can have several views in a DE, and you can have several ADO recordsets in a VB form. You'd just drop several ADO Data Controls on the form (and they'd be named, by default, ADODC1, ADODC2, and so on).

In **Figure 30.25**, I've sorted the Properties window with the Categorized tab, so that I can see the data properties all together and segregated from the other properties.

Clicking on the DataSource property will open a combo box that lists all of the available ADO recordsets—in this case, the combo box had only one entry, ADODC1. Then, click on Data Field. The combo box will be populated with all of the fields that are in the record source you specified. In this example, all of the fields in the tblCar table are available. If you created a different record source, say, by a SQL SELECT command, that resulted in a four-field result set, only those four fields would be displayed in the DataField property combo.

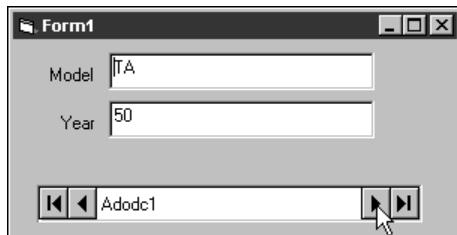


**Figure 30.25.** Set the *DataSource* and *DataField* properties of a text box to attach it to a specific field in a database.

Do the same for the Year text box, and save your form.

To see the Data Control in action, run the form by clicking the square button in the toolbar. You'll see the form as shown in **Figure 30.26**.

Admittedly, this isn't very fancy, but it gives you two essential abilities. The first is to navigate through a recordset. You can click the four buttons on the data control to move to the first, previous, next, and last records in the recordset. The other is to edit a value. This isn't immediately obvious, because there is no Edit or Save button, but the text boxes are indeed live. Change a value, move to another record, and voila!, your change has been committed.



**Figure 30.26.** You can use the intrinsic controls in the ADO Data Control on the form to navigate to the first, next, previous, and last records in the table.

## Revisiting the recordset idea

Okay, you're probably not going to pull this code out of this chapter and use it in an application. But it demonstrates the basic idea of an ADO recordset—the new way for Visual Basic applications to address data. A recordset, then, is just a cursor in Visual Basic clothing. It doesn't have to be the whole table, and it can contain data from more than one table as well.

But an ADO recordset is better than a cursor—it's actually an object, and as such has properties that you can manipulate and methods that you can call in order to work with the recordset. For example, the `AbsolutePosition` property is much like the `Recno()` function in Fox—it tells you which record in the recordset you're on. And the `MoveFirst` and `MoveNext` methods, for example, move the record pointer to the first record and the next record in the recordset, respectively. The syntax looks like this for a recordset named `adodcNewCustomers` (as opposed to the unhelpful nomenclature of “`adodc1`”):

```
adodcNewCustomers.Recordset.AbsolutePosition  
adodcNewCustomers.Recordset.MoveFirst  
adodcNewCustomers.Recordset.MoveNext
```

If you're thinking this all looks awful familiar, well, right you are. You already *know* all this, don't you?

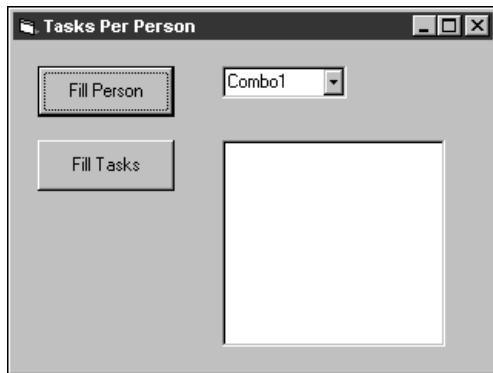
In summary, data is data is data. Connecting to it in Visual Basic using ADO is more involved than the `USE` command you're used to in Fox, but once you've made the connection, you can manipulate the recordset just as you work with a view in Fox.

## Real-world usage of ADO in VB

Now that you understand why you want to use ADO and how to get a connection started, I'll describe how to do a couple of real-world tasks with ADO.

### Populating a combo box and a list box

In this example, I'm going to use a simple form that contains two command buttons, a combo box, and a list box. Initially, the combo and list boxes will be empty. The first command button will populate the combo box with a list of distinct values from a table, and the second command button will use the value selected in the combo box to populate the list box with associated values. See **Figure 30.27**.



**Figure 30.27.** Our sample form to demonstrate hand-coded ADO.

Specifically, this form is digging data out of a “Tasks” table. This table is contained in a Microsoft Access 2000 .MDB named ATONCE.MDB. Each record in the table is a task that is assigned to an individual. The first command button, Fill Person, selects a list of distinct abbreviations for people to whom tasks are assigned as shown in **Figure 30.28**. (This way, tasks can be assigned to new people on the fly without having to first add the person elsewhere.)

Once the combo box is populated with people who have tasks assigned to them, the second command button, Fill Tasks, selects the value in the Task Description field for every record for that person, as shown in **Figure 30.29**. This actually is a lot of steps—you would probably automatically populate the list box upon selection of a new value in the combo box, but this is an example to show you how to perform different functions.

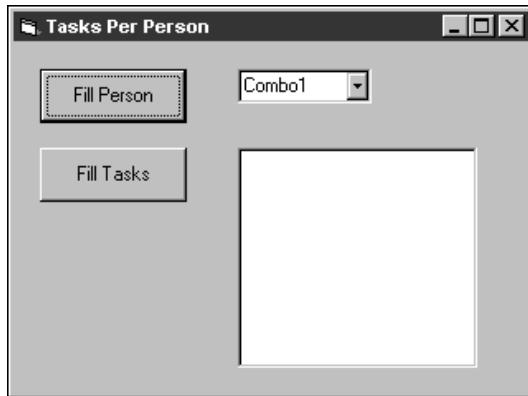
### Using form-level methods

It's generally good practice to keep your method code sparse—one sign of poor programming practice is Click() methods with hundreds or thousands of lines of code. So, even for this example, I created form-level methods (I'm sorry, in VB, they're called SUBS) that would be called from each of the command buttons. One sub would fill the combo box, and the other

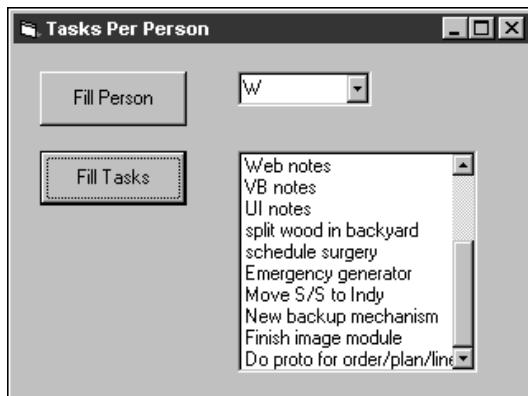
would fill the list box. Thus, when I decided I didn't need a separate Fill Tasks command button, I could just call the FillList() sub from the combo:

```
Private Sub cmdFillPerson_Click()
fillcbo
End Sub

Private Sub cmdFillTasks_Click()
filllist
End Sub
```



**Figure 30.28.** After clicking *Fill Person*, the user can select a specific person from the combo box.



**Figure 30.29.** After selecting a person in the combo box, the user can fill the list box with tasks for that individual.

## Filling the combo box

The code to accomplish this task—pulling one instance of each task owner from the Tasks table—is fairly straightforward. First, you create a pair of variable references: one for a connection and one for a recordset.

```
Dim objcn As New ADODB.Connection  
Dim objrs As New ADODB.Recordset
```

Next, assign values to the properties of the connection, like so:

```
With objcn  
    .CommandTimeout = 15  
    .ConnectionTimeout = 15  
    .Provider = "Microsoft.Jet.OLEDB.4.0"  
    .Open "data\atonce.mdb", "admin"  
End With
```

The Open method assumes that there is a directory called DATA underneath the directory that holds the program—in real life, you’d use a property to hold this value so that users could modify the location of the data as they desired.

Next, it’s time to populate the empty Recordset object, like so:

```
objrs.Open "select distinct cOwner from TASKS", objcn
```

This Open method uses the connection that I set up just a few lines of code earlier.

Once you have a recordset (remember, it’s just like a cursor in VFP), you can work with it as desired. For example, I scrolled through the recordset and added items from it to the combo box:

```
Do While Not objrs.EOF  
    Combo1.AddItem objrs!cOwner  
    objrs.MoveNext  
Loop
```

Then I used the MoveFirst method of the Recordset object to position the record pointer back on the first record of the recordset, and used that value to populate what you and I would call the display value of the combo box:

```
objrs.MoveFirst  
Combo1.Text = objrs!cOwner
```

Finally, I closed all of the objects because I don’t need them anymore:

```
objrs.Close  
Set objrs = Nothing  
objcn.Close  
Set objcn = Nothing
```

The entire method looks like this:

```
Sub fillcbo()
Dim objcn As New ADODB.Connection
Dim objrs As New ADODB.Recordset
With objcn
    .CommandTimeout = 15
    .ConnectionTimeout = 15
    .Provider = "Microsoft.Jet.OLEDB.4.0"
    .Open "data\atonce.mdb", "admin"
End With
objrs.Open "select distinct cOwner from TASKS", objcn
Do While Not objrs.EOF
    Combo1.AddItem objrs!cOwner
    objrs.MoveNext
Loop
objrs.MoveFirst
Combo1.Text = objrs!cOwner
objrs.Close
Set objrs = Nothing
objcn.Close
Set objcn = Nothing
End Sub
```

### Filling the list box

Once the user selects an item from the combo box, I'm going to fill the list box with the appropriate tasks for the selected individual. First, create a couple of objects for the connection and the recordset, and initialize another variable to hold the SQL command that's going to actually work on the Tasks table:

```
Dim objcn As New ADODB.Connection
Dim objrs As New ADODB.Recordset
Dim strSQL As String
```

Populate the Connection object just as was done for the combo box:

```
With objcn
    .CommandTimeout = 15
    .ConnectionTimeout = 15
    .Provider = "Microsoft.Jet.OLEDB.4.0"
    .Open "data\atonce.mdb", "admin"
End With
```

This part is new. Create a text string that represents the SQL statement to select the tasks (the cSubject field in the Tasks table) based on the value of the combo box. Then issue the Execute method of the connection using this string. You've now got a recordset populated just like in the combo box:

```
strSQL = "select cSubject from TASKS where cOwner = " _
& "'" & Combo1.Text & "'"
Set objrs = objcn.Execute(strSQL, adCmdText)
```

I've added a bit of fine-tuning here when filling the list box. Instead of just jamming the values from the recordset into the list box, I first cleared the items in the list box. Otherwise, the tasks from the new person would be added to the existing list of tasks:

```
List1.Clear  
Do While Not objrs.EOF  
    List1.AddItem objrs!cSubject  
    objrs.MoveNext  
Loop
```

Finally, close the objects as before:

```
objrs.Close  
Set objrs = Nothing  
objcn.Close  
Set objcn = Nothing
```

Here's the code for the entire method:

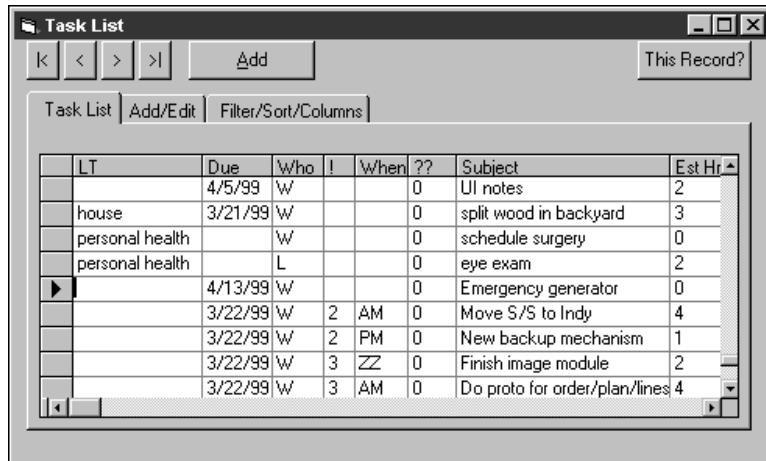
```
Sub filllist()  
Dim objcn As New ADODB.Connection  
Dim objrs As New ADODB.Recordset  
Dim strSQL As String  
With objcn  
    .CommandTimeout = 15  
    .ConnectionTimeout = 15  
    .Provider = "Microsoft.Jet.OLEDB.4.0"  
    .Open "data\atonce.mdb", "admin"  
End With  
strSQL = "select cSubject from TASKS where cOwner = " _  
    & "" & Combo1.Text & ""  
Set objrs = objcn.Execute(strSQL, adCmdText)  
List1.Clear  
Do While Not objrs.EOF  
    List1.AddItem objrs!cSubject  
    objrs.MoveNext  
Loop  
objrs.Close  
Set objrs = Nothing  
objcn.Close  
Set objcn = Nothing  
End Sub
```

As you can see, the big difference is in the syntax of creating a connection and populating a recordset, and in working with controls that operate a little differently. But the other things you're used to doing—working with relational tables and SQL—he, that's yesterday's news! Again, data is data is data. Connecting to it in Visual Basic using ADO, whether you use the controls or write the code yourself, is more involved than the USE command you're used to in Fox, but once you've got the connection made, you can manipulate the record set just as you work with a View in Fox.

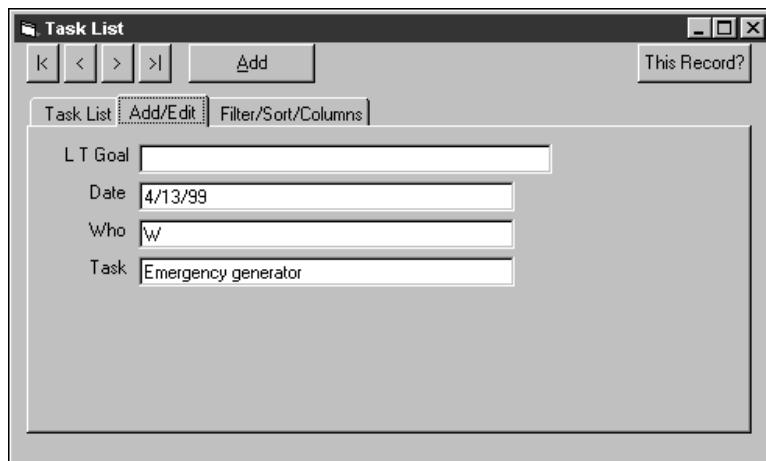
## Populating a grid

I'm using a page frame as my data-entry form. On the first tab, I've got the DataGrid control as shown in **Figure 30.30**.

**Figure 30.31** shows the second tab of the form—a place where the user can add or edit records. I've added a couple of nice touches to this tab. First, if you have an existing record highlighted in the Task List tab, clicking the Add/Edit tab displays the record in “file card” view as shown in Figure 30.31. In this sample form, I've not put all of the fields on the tab—just enough so you get the idea.



**Figure 30.30.** The Task List tab contains a Data Grid.



**Figure 30.31.** Clicking the Edit tab allows the user to see the entire record in a “file card” format.

If, on the other hand, you click the Add button, you're automatically moved to the Add/Edit tab, and have a blank record at your disposal for entering new data. When you tab out of the last field, focus is returned to the Task List tab.

The third tab, Filter/Sort/Columns, isn't really active yet, but eventually it will be used to allow the user to customize the view of the grid—which records (that would be a filter, right?), the order of the records (Sort), and which columns to display—and in which order to display them (Columns).

In addition, the four navigation buttons in the upper left corner allow you to move through the grid, a record at a time, or to the first or last records, while the This Record? button in the upper right simply displays a message box with the number of the current record. Okay, maybe you would find this on a mission-critical application that spans four continents, but it's useful for our discussion.

### Setting up the data control

What you don't see on the form, of course, are the other two controls—the data controls I used to populate various parts of the form. Therefore, in **Figure 30.32**, you see the form in design mode, with the two data controls in the lower right corner of the form. I'll just focus on one of the data controls now—the one that fills the data grid on the first tab of the page frame. I also show the Property Pages for ADODC1. (Right-click on the data control and select the ADODC Property Pages menu to open the Property Pages dialog.)

In Figure 30.32, I show the General properties for the data control that will fill the grid. The connection string is a standard Access 2000 .MDB (it kills me every time I have to type that!), and the data source points to the ATONCE.MDB that's located in the data directory underneath the source code for this project. Note that Visual Basic will try to include the entire path for the data source—which, of course, plays hell for portability.

Pointing to an .MDB, of course, doesn't do any more good than pointing to a .DBC in Visual FoxPro. You need to describe what part of the .MDB you want to get at. In **Figure 30.33**, I show the RecordSource properties—the Command Type is a table, and the specific table inside the .MDB is the Tasks table.

### Adding controls to your toolbox

Now, look away from the screen (or the paper, if you printed this out), and then come back to this chapter. The next screen shot is also a Property Pages dialog, but it's not of the ADODC1 data control—it's of the DataGrid control. Before I drill into each of the properties, however, I should explain how I got the DataGrid on the form at all. (Kind of like explaining where the Start button is before explaining how to shut Windows down, right?)

In the VB toolbox, you can right-click and select the Components menu option to open the Components dialog as shown in **Figure 30.34**. Scroll about four bazillion rows down in the list box on the Controls tab, and you'll find the Microsoft ADO Data Control 6.0 (SP3) and the Microsoft DataGrid Control (SP3). Note that they are both OLE DB controls. This is good—you'll want to stay away from data-oriented controls that aren't OLE DB because they're as my daughters would say, like, so five minutes ago (or, in our vernacular, "old fashioned"). These days, the two terms are basically the same, of course. Be sure to check the check box to the left of each control you want to appear on the toolbox; simply highlighting one and clicking Apply isn't good enough.

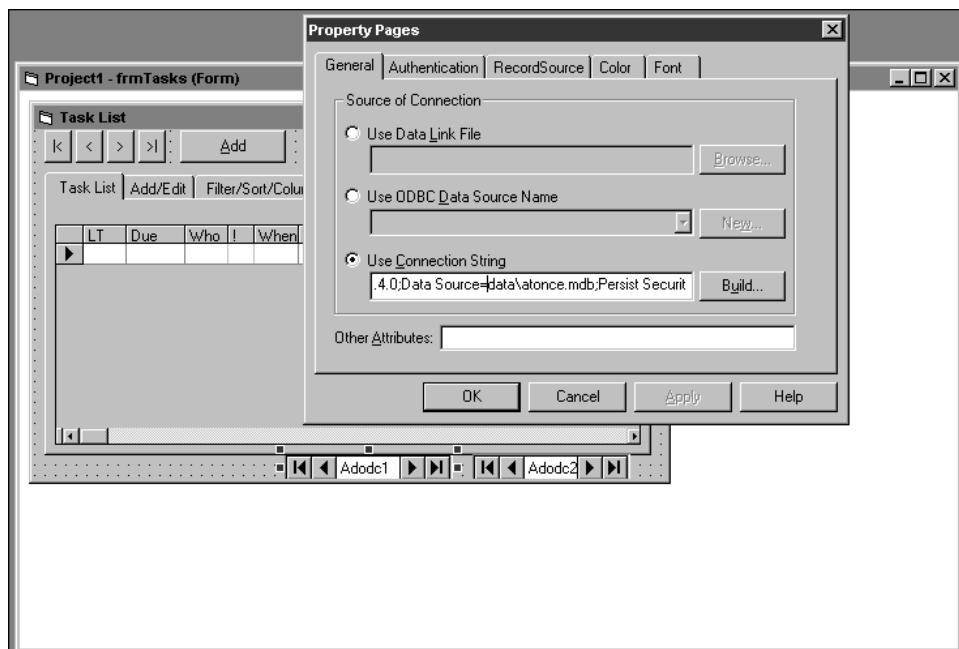


Figure 30.32. The General tab of the ADODC data control shows the connection string.

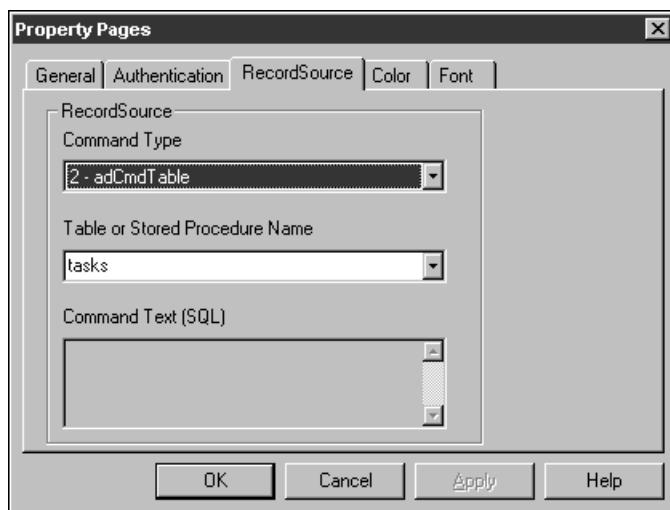
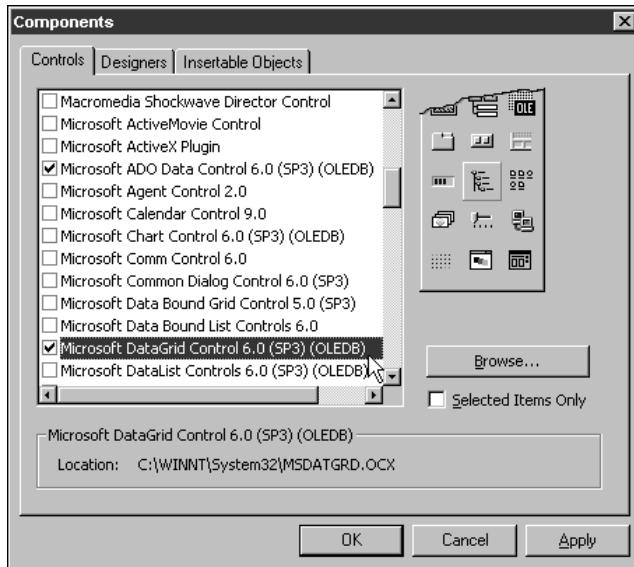


Figure 30.33. The RecordSource for the ADODC1 data control is the Tasks table in ATONCE.MDB.



**Figure 30.34.** Selecting good ActiveX controls from the Components dialog.

Once these controls are on the toolbox, of course, you can select one and drop it onto your form. (Be sure to select the page frame control and select the appropriate page so that you drop your control on the page you want it to appear on.) Finally, right-click the DataGrid control and select the Properties menu option.

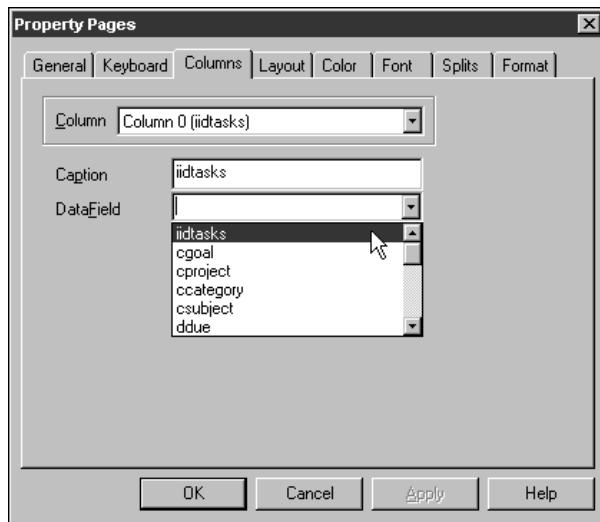
### Setting up the DataGrid control

I could spend three or four chapters just covering the various options in this control, and indeed, I'd like to. But it would get pretty boring because there's lots of other good stuff to cover as well. So how about just the high spots—you can dig in further if you want to yourself.

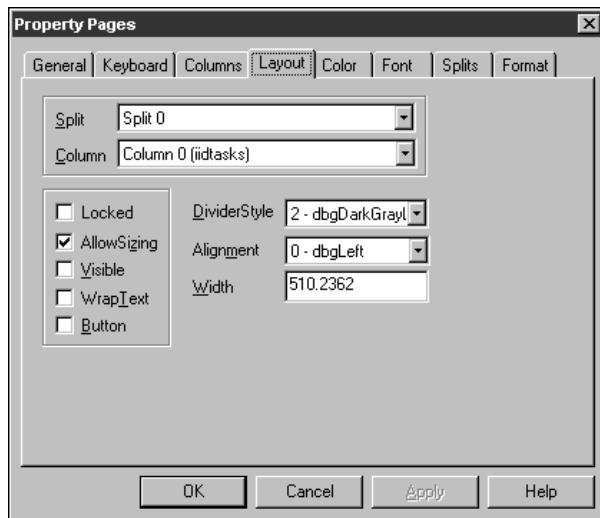
The only real customizing needed before you dive into this new set of property pages is to identify the DataSource property for the DataGrid control. Select the Properties window (not the pages!), order the properties by Category, and select DataSource property. You'll see a combo box populated with each of the recordsets that you've added to the form. Pick the appropriate one (if you name your recordsets with something more useful than "ADODC1" and "ADODC2", it will be easier to remember which one to use.)

The Columns tab, as shown in **Figure 30.35**, allows you to control which field is going to land in which column. The trick here is to remember it's not which column is going to show up—you get to filter the columns for display in a moment. Anyway, I've tied the Primary Key field, iidtasks, to the first column (and as with every Visual Basic index, the index starts with zero). Where do the contents of the DataField combo come from? Other than magic, that is? They come from the ADO DataControl, and because I specified "Table" in the RecordSource property, every field in the Tasks table will be available. I could have specified a cursor instead of the entire table, and then just the contents of the cursor would have been available.

The good stuff comes next—in the Layout tab. You can set a number of properties for each column in the DataGrid, including Locked, AllowSizing, Visible, and WrapText. I simply set Visible off, as shown in **Figure 30.36**, and now there's no confusing Primary Key field displaying in the DataGrid.



**Figure 30.35.** The DataField combo is populated with field names from the ADO Data Control recordset.



**Figure 30.36.** Clicking on the Edit tab allows the user to see the entire record in a “file card” format.

You can set about four bazillion more properties in the Splits tab (a split is similar to the Excel splitter) as shown in **Figure 30.37**. I use this tab to mention a beef about Visual Basic and MSDN Help. You see 11 controls on this tab, right? So, suppose you wonder what “ScrollGroup” means. You click on Help, thinking that there might, at the very least, be a small, helpful paragraph describing what “ScrollGroup” is, right?

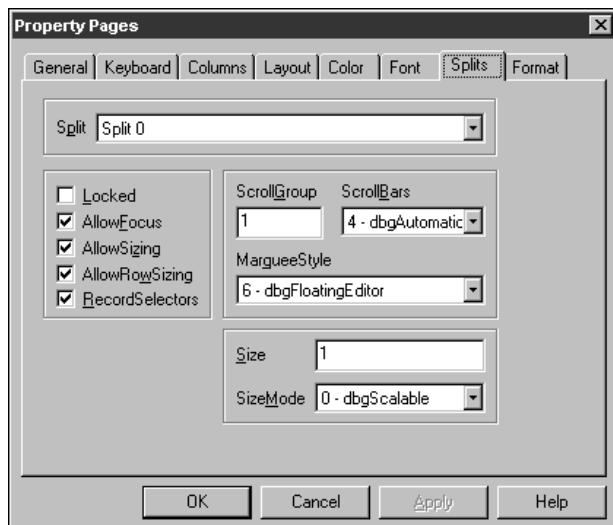
Heck, no. You get a description (after following four hyperlinks) for the “split” property of an object, and it has nothing at all to do with the “ScrollGroup” control on this tab. Pardon my French, but what a crock! I’d rather they just rip the Help button off the dialog completely, rather than provide useless and misleading information like this. Okay, end of rant. But be prepared to experiment, because you won’t find out how these things work any other way.

The last tab on this page, Format, provides you with the ability to control how the data in each column is formatted, as shown in **Figure 30.38**.

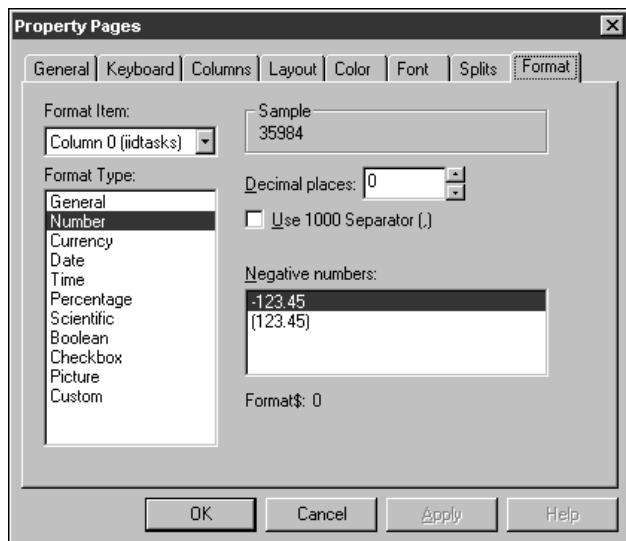
Okay, now you’ve got a quick run-down on the DataGridView control.

If you’re a Visual Basic developer, the mind just reels with the flexibility that this control provides you. As a Fox developer, however, you’ve probably been very polite, quietly sitting in the back, but just aching to raise your hand: “How do you change the data source underlying the grid? In Fox, you just do a new SELECT and then refresh your grid.” Or, “How can you color each row a different color?” Or, “How do you put check boxes in the third row?”

The answer will probably be unsatisfying, but just hold your horses for a minute. The answer is: “You buy another grid control that provides the functionality you want.” That’s how VB was designed, after all. We’re spoiled because we’ve got so much functionality right in the box, but on the other hand, we’ve also had to be satisfied, since a lot of ActiveX controls don’t work well with VFP. So the situation is kind of like a pair of golden handcuffs, right?



**Figure 30.37.** The Splits tab has 11 controls, none of which has any information in Help.



**Figure 30.38.** The Format tab allows you to determine how data in a column will be displayed.

You can, of course, programmatically change all of these properties, and if you want, you can even repopulate the contents of the grid by creating your ADO recordset in code, and then refreshing the grid. It seems to me like it's more work than in VFP, but I venture that's due more so to familiarity than to fact.

As you can see, it's not a big deal to connect a rather complex control to an ADO data control. The bigger job is to decide which control you want to use, given the set of requirements that you have (and which, likely, continue to change as you develop the app).

## A quick ADO command reference

So, Bunky, you don't want to experiment with learning ADO recordset syntax one command at a time? You'd like a quick reference of what's available? In this section, I'll show you what ADO keywords are available (at least in this week's version), how some of them compare to VFP, and describe what you'd use some of them for.

As I discussed earlier, a recordset (or, more accurately, a Recordset object, since it's got properties and methods attached to it) contains either the entire set of records from a table or the results of a command like a SQL SELECT. It's fundamentally just like a Visual FoxPro cursor except for those extra goodies—properties and methods. But you work with it the same way: navigate through the set of records and perform operations on the current record.

It's important to note that not all providers support the entire ADO syntax listed below. You can use the Supports() method to determine details.

## Properties

|                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| AbsolutePage     | Returns the page on which the current record resides. This may not always be available, depending on the provider.                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| AbsolutePosition | Specifies the “ordinal position” of the current record. You and I know this as “the record number.”                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| ActiveConnection | Returns the name of the connection object to which the current recordset object belongs.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| BOF, EOF         | Indicates that the current record position is before/after the first/last record in a Recordset object. This is slightly different than Fox in that you can think of ADO as having two phantom records—one at the beginning of the file and one at the end. Once you move onto one of these phantom records, BOF or EOF is set to True and there is no “current record.”                                                                                                                                                                                                                      |
| Bookmark         | <p>You know how you have to go through a bunch of code like the following in order to save the position of the record pointer so that you can return to it?</p> <pre>if eof()   m.liCurRec = reccount() else   m.liCurRec = recno() endif &lt;do some fancy stuff like moving around&gt; go m.liCurRec</pre> <p>With ADO, you can set a bookmark that serves the same purpose, as “m.liCurRec.” This is not available in all recordsets.</p>                                                                                                                                                  |
| CacheSize        | Returns the number of records in a Recordset object that are locally cached.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| CursorLocation   | Sets or returns the location of the cursor engine: Client or Server.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| CursorType       | <p>Sets or returns the type of cursor used. Options are ForwardOnly, Keyset, Dynamic, and Static. Each type of cursor trades off between performance and flexibility.</p> <p>ForwardOnly: Same as static except allows only forward scrolling.</p> <p>Keyset: Same as a dynamic cursor except Adds by other users aren't visible.</p> <p>Dynamic: The most flexible of cursors—allows you to see modifications made by other users and all types of navigation.</p> <p>Static: A static set of records that you would typically use for reporting. Allows forward and backward scrolling.</p> |
| EditMode         | Indicates the edit status of the current record—much like the buffer in VFP.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| Filter           | Allows you to set or determine a filter for a recordset.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| LockType         | Indicates the type of locks placed on records during editing. You can choose between ReadOnly, Pessimistic, Optimistic, and BatchOptimistic. Hmm, sound familiar?                                                                                                                                                                                                                                                                                                                                                                                                                             |
| MarshalOptions   | <p>This is a bit advanced for a “quick-reference” so I'll quote directly from the help file:</p> <p>When using a client-side (ADOR) Recordset, records that have been modified on the client are written back to the middle-tier or Web server through a technique called <i>marshaling</i>, the process of packaging and sending interface method parameters across thread or process boundaries. Setting the MarshalOptions property can improve performance when modified remote data is marshaled for updating back to the middle-tier or Web server.</p>                                 |

**Properties, continued**

|                     |                                                                                                                                                                                                                                       |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| MaxRecords          | Much like when you specify how many records should be retrieved in a view, this allows you to limit the number of records placed in a recordset. The default value is 0, which means all desired records are placed in the recordset. |
| PageCount, PageSize | Tells you how many pages of data are in the recordset and what the page size is. Much like AbsolutePage, this might not be supported by the recordset.                                                                                |
| RecordCount         | Yup, just like Reccount() in VFP. Well, not exactly. There are some performance issues here—see online help for the details.                                                                                                          |
| Source              | Indicates the underlying source for a Recordset object. Choices are Command object, SQL statement, table name or stored proc.                                                                                                         |
| State               | Indicates whether an object is open or closed.                                                                                                                                                                                        |
| Status              | Indicates the status of the current record as far as batch updates and other group processes go. See online help for a long list of possible values.                                                                                  |

**Methods**

|              |                                                                                                                                                                                                                                                                                                                                                                     |
|--------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| AddNew       | Creates a new record for an updateable Recordset object.<br><br><code>rs.addnew fields, values</code>                                                                                                                                                                                                                                                               |
| CancelBatch  | Cancels a pending batch update.                                                                                                                                                                                                                                                                                                                                     |
| CancelUpdate | Cancels any changes made to the current record or to a new record prior to calling the Update method. Also discards a newly added record. Can't undo changes after you call Update unless the changes are part of a transaction or a batch update.                                                                                                                  |
| Clone        | Creates a duplicate Recordset object from an existing Recordset object. Returns a Recordset object reference.<br><br><code>Set rsDupe = rsOriginal.clone()</code>                                                                                                                                                                                                   |
| Close        | Closes a Connection object or a Recordset object. Closing a connection also closes active Recordset objects associated with the connection.                                                                                                                                                                                                                         |
| Delete       | Deletes the current record or a group of records.<br><br><code>rs.delete RecordsToDelete</code><br><br>RecordsToDelete is a constant that is equal to adAffectcurrent (only delete the current record) or adAffectGroup (deletes the records that are in the current filter). Obviously, you must set the Filter property first.                                    |
| GetRows      | Places multiple records of a recordset into an array.<br><br><code>Array = rs.getrows(rows, start, fields)</code><br><br>After you call GetRows, the record pointer is placed at the first record that wasn't put into the array, or at EOF if there are no more records.                                                                                           |
| Move         | The doc says "Moves the position of the current record in a recordset object" but this is awkwardly worded—to you and me, it means "move the record pointer to a new record."<br><br><code>rs.move NumberOfRecords</code><br><br>If you move before the first record, the current record is before the beginning of the file, and BOF is set to True. Same for EOF. |

***Methods, continued***

|                                                      |                                                                                                                                                                                                                                                                                                                         |
|------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| MoveFirst,<br>MoveLast,<br>MoveNext,<br>MovePrevious | Moves to the specified record                                                                                                                                                                                                                                                                                           |
| NextRecordset                                        | This one requires a bit of explanation. You can create a series of recordsets with a compound command statement, such as:<br><br><code>"select * from TABLE1; select * from TABLE2; select * from TABLE3"</code><br><br>This method will close the TABLE1 recordset and open the TABLE2 recordset in turn.              |
| Open                                                 | Opens a recordset.                                                                                                                                                                                                                                                                                                      |
| Requery                                              | Similar to requerying a combo box or a list box, this method will refresh a recordset by requerying the underlying source database.                                                                                                                                                                                     |
| Resync                                               | Similar to requery but does not handle adds or deletes in the underlying source database. In other words, if a record was added to the underlying database, Resync will not see that record and will not include the record in the recordset.                                                                           |
| Supports                                             | Determines whether a specified Recordset object supports a particular type of functionality. The parameter passed to the Supports() method identifies a feature that may or may not be supported by the recordset in question.                                                                                          |
| Update                                               | Saves changes made to the current record of a recordset.                                                                                                                                                                                                                                                                |
| UpdateBatch                                          | Writes all pending batch updates to disk. Used when modifying a recordset in batch update mode. Instead of calling Update for each individual record in a recordset, you can cache all changes to a recordset and then send the entire set of changes back to disk with this command. Sorta like using table buffering. |

The more things change, the more they stay the same. ADO's heritage—the development team is largely made up of people from the Fox data engine crew—is evident throughout. Concepts that are new and strange to VB developers are yesterday's news for Fox developers. But that's not the only place where you've got a leg up.

## What's this business about Visual Basic classes?

I believe it was Abraham Lincoln who posited the question, "If you call a tail a leg, how many legs does a dog have?" The answer, then, was, "Four. Calling a tail a leg does not make it one." Unfortunately, some idiot in Redmond hadn't heard this one, and decided to use the term "classes" in order to name Visual Basic's implementation of templates. To close out this chapter, I'm going to discuss, at a high level, what Visual Basic classes are, and why you shouldn't bother with them.

VB classes are templates for creating visual interfaces or non-visual processes. They're by default created and stored on drive C, much like the VFP Component Gallery. (I'll get into why putting development tool data on drive C is such a *stupid* idea some other time, but it's nice to know that Fox isn't the only development team that does this.) VB classes don't support inheritance. Period.

You create a class for an object, and then you can create objects in your application using that class. If you change the class, however, the objects you created from the original version of that class do not change. Period.

While you and I are grimacing, evidently this is a big freaking deal to VB developers. To listen to a group of VB developers, you'd have thought they just learned how to turn lead into

gold. Actually, if I get off my “Fox Rules” pedestal for a moment, it’s easy to understand. Remember how we thought we were hot stuff when we got TEN work areas? When we could do light-bar menus? Or when we could use GENSCRNX to enhance the screen generation process? So we’ve got OOP now. I’m sure in 30 years we’ll all be averting our eyes, digging our toes in the dirt, embarrassed to admit how exciting we were about “Object Oriented Programming” when it’s become a minor stepping stone in the curriculum from fourth grade to fifth.

Given where VB programmers were with version 4.0, classes were a major step forward. Until classes were introduced, VB developers literally created every application from scratch. While we were building 2.x frameworks or using tools like FoxExpress or MaxFrame, they were hand-coding applications from line one, and their most advanced RAD technique was “Ctrl+C.” Having the ability to create templates that can be reused was indeed a major step forward, not only in the coding process, but also in getting developers who formerly coded by the seat of their pants to start planning—yes, to start thinking about designing their apps ahead of time.

You might have detected a bit of anger here. And you’re right. First of all, this is the second time I’ve written this section. Microsoft Word 2000 took a hike south on me, trashing my backup in the process, just as I was going through the final edits a week ago. I think it’s criminal that version 9 of a silly word processor is still as unreliable as Word is. Sure, we’ve got fancy menus that change according to your use of them, and there are 17 different ways to create a Web page from your Word document, but you still can’t depend 100% on being able to save a file. I guess this approach to developing software—shove it out the door before it’s ready—will keep us employed for many years. Anyway, I digress.

I’m also angry—actually, much angrier—at two other aspects of this issue. The first is just personalities; it’s been a bit tough to stomach folk who strut around like cock of the walk: “We’ve got CLASSES! How about us! Ain’t we something!” while ignoring that the rest of the world has had the real thing for years. (Of course, as the saying goes, let those who are without sin throw the first stone. We Fox developers have had our nose up in the air for quite a while and other developers are pretty darn tired of it as well. But it’s hard to be humble when you’ve got a tool as great as VFP, isn’t it?)

But this has polarized the VB and VFP communities, and that’s too darn bad. We’ve got a lot of knowledge about design, OOP, and databases that we could share with the VB community. And the VB guys could teach us a ton of stuff about working with Windows, ActiveX, and other Microsoft technologies that we’ve generally tried to stay away from. If you’re not reading *VBPJ* right now, you’re behind the times, for example. I’m afraid that the average developer in both camps has had their bridges burned by the animosity between the two groups.

The other thing that’s really aggravating about this “class” business is that it has brought out some of the worst writers on the planet and plunked them into the VB community. I’ve yet to see a lucid description of what classes are and how they work in anything I’ve read, and that’s a shame because there are a couple of geniuses in the VB community. I’ll read anything that Dan Appleman writes, for example, whether I’m interested in the topic or not, just because he’s fun and engaging.

C programmers have “Obfuscated Code Contests” where you show someone else your code, and dare them, “I bet you can’t tell me what THIS does!” I get the same feeling when VB gurus write about classes. Here’s my favorite quote:

“The classes we can create in VB support the concepts of polymorphism and encapsulation, but not inheritance. That’s just fine. While inheritance can be useful when structuring collections of classes, it can be a major source of bugs and its implementation requires the use of code that is difficult to understand.”

—John Connell, *Beginning VB6 Database Programming*, WROX Press.

Yes, this is French for “You’re too stupid to use inheritance correctly, so it’s just as well that it’s not in VB.” And then—here’s the corker—he dedicates the next 65 pages in the book to his first example on using classes. SIXTY FIVE pages for an introductory example? We’ve covered a complete introduction to the language in that amount of space. But this is kind of common. I’ve seen the same approach, albeit somewhat more moderate, in several other books.

So what’s the deal? Is this topic just too hard? Is the mechanism too difficult to demonstrate in anything less than 25,000 words? Or is it that they don’t understand it? Or is it that they’re embarrassed that the implementation is half-baked, and they’re trying to distract you from the fact? I dunno. If you’ve come across a VB book that does this justice, please let me know!

Well, I’ve tried to build classes in VB myself, and it *is* hard. There are a lot of steps—just like in VFP—to get your classes set up, and then to create objects based on those classes. And while I guess it’s better than hand-coding, it’s still a lot of work considering that the first time you change something, you have to start throwing out your previous work. Three steps forward, two steps back.

I can’t believe that this is going to be as good as it gets in VB—I have to think that the object model will improve, and that some day, VB will get true inheritance as well. But until then, well, “it’s too hard” and I think I’ll wait.



# Index

Tip: The Fundamentals online help (.CHM) file includes a full text search – you can search on any word or phrase in the book.

- "Command-line switches", 452
- #DEFINE, 220
- #INCLUDE, 390
- #UNDEF, 220
- .CDX, 72
- .DBF, 72
- .FPT, 72
- .IDX, 72
- .MEM files, 80
- 1999 Software Developer's Guide, 313
- Abstract class, 262
- Accessing an existing database, 93
- Accessing an existing table, 82
- Accessing multiple databases, 94
- Accessing multiple tables, 84
- Accessing the project object, 505
- Accessing the project object, Application object, 505
- Accessing the project object, File object, 509
- Accessing the project object, Files collection, 509
- Accessing the project object, Project object, 507
- Accessing the project object, Projects collection, 507
- Accessing the project object, Server object, 510
- Accessing the project object, Servers collection, 509
- ACTIVE WINDOW, 543
- ActiveProject, 507
- ActiveX, 599
- ActiveX, DLL hell, 611
- ActiveX, How do I put them into my development environment?, 605
- ActiveX, How do I ship them with an application?, 610
- ActiveX, How do I use them in an application?, 607
- ActiveX, What if it doesn't work?, 610
- ActiveX, What is an ActiveX control?, 599
- ActiveX, Where-and how-do I get them?, 600
- ADATABASES(), 94
- Add Catalog, 497
- Add Table, 97
- Add(), 401
- Adding a batch of records from another source, 117
- Adding records interactively, 115
- Adding records programmatically, 115
- AddItem, 247, 250
- ADO, 676
- ADO, A quick command reference, 768
- ADO, ADO errors, 695
- ADO, Command object, 695
- ADO, Connection object, 694
- ADO, Dealing with constants, 696
- ADO, Errors collection, 695
- ADO, Fields collection, 694
- ADO, Getting started, 680
- ADO, History of the Microsoft data access strategy, 677
- ADO, Object model, 680
- ADO, OLE DB, 677
- ADO, Parameters collection, 695
- ADO, Recordset object, 694
- ADO, What is ADO?, 678
- Advanced Object Oriented Programming, 487
- Advanced Object Oriented Programming with Visual FoxPro 6.0, 259, 291, 494
- AfterBuild event, 515
- Aggregate functions, 129
- ALTER TABLE, 107
- Always on Top, 211
- AND, 21
- Appearance, 158
- APPEND BLANK, 143
- Application class, 264
- Application foundation, 347
- Application object, 505
- Arrays, 23
- Attribute, 70
- A\_EVENT, 462
- A\_HELP, 462
- A\_PROC, 461
- A\_USER, 461
- BackColor, 228
- Bar #, 178
- Beautify, 156
- BeforeBuild event, 514
- Binary data, 74
- BOF() EOF() and the Phantom Record, 109
- BoundColumn, 414
- BROWSER.DBF, 496
- Build Options, 165
- BUILDER property, 562
- BUILDER.DBF, 560
- BUILDERD, 565
- Builders, 557
- Builders, Creating a user interface for a builder, 563
- Builders, Data-driving your builder-building process, 565
- Builders, Setting up your own builder in a BUILDER property, 562
- Builders, Setting up your own builder in BUILDER.DBF, 560

Builders, Visual FoxPro Builder technology, 558  
BUILDFILES, 463  
Building commands functions and expressions, 17  
By List, 219  
ByWho parameter, 455  
C.J. Date, 69  
Call Stack option button, 524  
Call Stack window, Show call stack indicator, 524  
Call Stack window, Show call stack order, 524  
Call Stack window, Show current line indicator, 524  
Camel case, 204  
Candidate key, 71  
CD(), 150  
Character binary, 75  
Character fields, 73  
Check Box, 10  
Checking for \_screen.activeform, 427  
Child record, Deleting, 426  
Child record, Editing, 425  
Class, 261  
Class Browser, 289  
Class Browser, Starting, 289  
Class Browser, Using, 289  
Class Designer, 288  
Class hierarchy, 262  
Class libraries, Creating, 350  
Class library, 267  
Classy Components, 477  
CLEAR EVENTS, 323  
Click, 231  
Click event, 205  
Click method, 205  
Client/Server, 24  
CLOSE DATABASE, 98  
Closing a table, 86  
Code Complete (Microsoft Press), 555  
Code pages, 75  
Collection, 507  
Color Palette toolbar, 302  
Colors, 25  
ColumnCount property, 253  
Combo Box, 10  
Command and function, 13  
Command button, 9, 236  
Command Button Group, 263  
Command option, 177  
Command window, 6  
Command window, Context menu, 15  
Command window, Properties, 17  
Command window, Using, 14  
Commands, 17  
Comment, 156  
Common libraries, 457  
Common User Access, 314  
Compiling a program, 152  
Component Gallery, 487  
Component Gallery, ActiveX Catalog node, 491  
Component Gallery, Adding your own components, 500  
Component Gallery, Arranging components, 497  
Component Gallery, Catalog data, 494  
Component Gallery, Catalogs node, 490  
Component Gallery, Catalogs vs. folders, 490  
Component Gallery, Favorites node, 491  
Component Gallery, Global settings, 496  
Component Gallery, Loading, 488  
Component Gallery, My Base Classes node, 491  
Component Gallery, Using a component in your project, 502  
Component Gallery, Visual FoxPro Catalog node, 493  
Component Gallery, What are catalogs?, 489  
Component Gallery, What's in a catalog?, 489  
Composite key, 71  
Configuring development application directory structures, 455  
Consolidating code in a generic method, 382  
Constants, 390  
Container, 195, 222  
Context menu, 8  
Context menus, Creating, 189  
Control, 195  
Control and container classes, 277  
Control classes, 275, 373  
Controls, 8  
Controls, Check Box, 243  
Controls, Command Button, 236  
Controls, Command Button Group, 238  
Controls, Common events, 231  
Controls, Common properties, 228  
Controls, Edit Box, 243  
Controls, Event firing, 232  
Controls, Grid, 252  
Controls, Hyperlink, 240  
Controls, List Box and Combo Box, 245  
Controls, Option Group, 244  
Controls, Spinner, 251  
Controls, Text Box, 240  
Controls, Timer, 238  
ControlSource property, 251, 253  
Coverage Profiler, 567  
Coverage Profiler, Basic Reports, 570  
Coverage Profiler, context menu, 572  
Coverage Profiler, Customizing with Add-Ins, 579  
Coverage Profiler, Find, 577  
Coverage Profiler, Open, 579  
Coverage Profiler, options, 574  
Coverage Profiler, Quick Start, 567  
Coverage Profiler, Save, 579  
Coverage Profiler, Statistics, 577  
Coverage Profiler, toolbar, 572  
CREATE, 89  
Create project, 161

CREATE TABLE, 102  
Creating a program with the editor, 149  
Creating a real form from your base class, 359  
Creating a table, 91  
Creating an application-level non-visual class, 341  
Creating an index tag interactively, 89  
Creating an index tag through the Table Designer, 89  
Creating OLIB, 430  
CURDIR(), 150  
Currency fields, 75  
Data files, 458  
Data handling, 392  
Data records, 73  
Data sessions, 208  
Data structures, Visual FoxPro, 72  
Data, Application specific, 458  
Data, Modifying, 115  
Data, Saving, 120  
Data, Working with, 109  
Database Designer, 96  
Databases, 70, 72, 79  
Databases, Accessing and existing database, 93  
Databases, Accessing multiple, 94  
Databases, Manipulating, 93  
Databases, Viewing the contents of, 94  
Date fields, 74  
DateTime, 35  
Datetime fields, 75  
DBASE, 3  
DBC(), 94  
DDE, 36  
Debugger, 521  
Debugger, Call Stack window, 538  
Debugger, Choosing a debugging frame, 522  
Debugger, Configuring, 521  
Debugger, Configuring windows, 523  
Debugger, Debug Frame menu, 545  
Debugger, Debug Output window, 540  
Debugger, Event tracking, 548  
Debugger, Locals window, 538  
Debugger, Log Debug Output check box, 525  
Debugger, Output option button, 525  
Debugger, SET DEBUGOUT TO, 525  
Debugger, The five debugger windows, 527  
Debugger, Toolbar, 543  
Debugger, Trace window, 527  
Debugger, Watch window, 536  
Debugging, 37, 550  
Debugging, Developer-generated errors, 553  
Debugging, Types of misbehavior, 550  
DEBUGOUT, 541  
Default behavior, 237  
Default directory, 443, 447  
DELETE, 145  
Delete(), 407  
DeleteMark, 253  
Deleting records interactively, 117  
Deleting records programmatically, 117  
Destination, 140  
Destroy, 231  
Developer tools, 457  
Developer Utilities, 447, 464  
Developer Utilities, .CHM files, 467  
Developer Utilities, Array Browser, 465  
Developer Utilities, CodeMine, 471  
Developer Utilities, DBI Technologies ActiveX controls, 472  
Developer Utilities, DevHelp, 465  
Developer Utilities, Eraser, 470  
Developer Utilities, FoxAudit, 473  
Developer Utilities, Foxfire!, 473  
Developer Utilities, HackCX, 466  
Developer Utilities, HTML Help Builder, 474  
Developer Utilities, INTL, 475  
Developer Utilities, Mail Manager, 476  
Developer Utilities, Mere Mortals Framework, 476  
Developer Utilities, OpenAll, 465  
Developer Utilities, QBF Builder, 478  
Developer Utilities, SoftServ's Command Window Emulator, 468  
Developer Utilities, Stonefield Database Toolkit, 478  
Developer Utilities, Stonefield Query, 479  
Developer Utilities, Stonefield Reports, 480  
Developer Utilities, Visual FoxExpress, 480  
Developer Utilities, Visual MaxFrame Professional, 482  
Developer Utilities, Visual Web Builder, 482  
Developer Utilities, West Wind Web Connection, 483  
Developer Utilities, WW, 464  
Developer Utilities, xCase, 486  
Developer Utilities, Z, 465  
Developer vs. user requirements, 452  
Developer-only controls, 436  
DisabledForeColor, 229  
Display Timer Events check box, 527  
Display Value property, 250, 251  
DISTINCT, 142  
DISTRFILES, 463  
Distributing applications, 647  
DLL hell, 611  
Document processor, 315  
Documentation, 460  
DODEFAULT(), 284, 421  
Domain, 70  
DOS search path, 448  
DoSets(), 339  
Drew Speedie, 482  
Driving an application with a menu, 173  
Drop-down menu, 8  
E.F. Codd, 69  
Edit Box, 9  
Edit menu, 317

Editing records interactively, 119  
Editing records programmatically, 119  
Effective Techniques for Application Development with VFP 6, 117, 418  
Enhancing the main program, 323  
Enhancing your form class with toolbars and menus, 365  
Entity, 70  
Environment - Disk, 38  
Environment - General, 37  
Environment - Keyboard, 38  
Environment - Monitor, 38  
Environment - Network, 38  
Environmental combo box, 522  
Error, 231  
Error handling, 39  
Escape behavior, 237  
Event handler, 322  
Event tracking, 548  
Events, 195  
Exclusive vs. Shared use, 83  
Execute Selection, 153  
Expectations, 550  
Expressions, 21  
F1 Technologies, 480  
Field, 70  
Field lists, 124  
Field mapping, 431  
Fields, 73  
File information, 40  
File name manipulations, 39  
File selection, 41  
Find, 155  
FLUSH, 121  
Focus and containers, 227  
Font, 42, 155  
ForeColor, 228  
Foreign key, 71  
Form classes, 268, 354  
Form Controls, 195  
Form Designer, 195, 209  
Form Designer, Code window, 214  
Form Designer, Form controls toolbar, 214  
Form Designer, Form designer toolbar, 214  
Form Designer, Include files, 220  
Form Designer, Layout toolbar, 195, 216  
Form Designer, Menus, 216  
Form Designer, Properties window, 195, 210  
Form Designer, Referencing objects, 221  
Form Designer, Tab order, 218  
Form properties, Changing through user input, 200  
Forms menu, 317  
Forms, Calling methods from objects on the form, 203  
Forms, Creating and Running, 197  
Forms, Integrating with data, 206  
Forms, Passing object references, 415  
Forms, Passing parameters to a form, 414  
Forms, Returning values from a form, 415  
Forms, Setting tab order, 428  
Forms, Terminology and tools, 195  
Forms, The big picture, 193  
Forms, Types of, 349  
Fox Foundation Classes (FFC), 491  
FOXPRO.H, 390  
FOXUSER, 225  
FULL JOIN, 134  
Functions, 20  
GATHER, 120  
General fields, 75  
Generate process, 200  
GENMENUX, 188  
GETFLDSTATE(), 401  
GO, 112  
Go To, 155  
GotFocus, 231  
Grid, 11  
GridLines properties, 253  
GROUP BY, 130  
GUID, 671  
Hackers' Guide to Visual FoxPro 6.0, 227, 451  
Handling deleted records: recalling and packing, 117  
HAVING, 143  
Header file, 220  
Header record, 73  
Help, 42  
Help menu, 319  
HTML Help, 621  
HTML Help, Creating using HTML Help Workshop, 625  
HTML Help, Including a .CHM file with your VFP application, 644  
HTML Help, What is HTML?, 621  
HWCTRL62, 373  
HWCTRL62.VCX, 350  
Hypotheses, Generate, 554  
Hypotheses, Test, 555  
IDE, 7  
Include file, 390  
Increased cohesion, 431  
Indenting, 158  
Indexes, 76  
Indexes and Rushmore, 79  
Indexes, Creating, 88  
Indexes, Index file, 78  
Indexes, Index file (for order by Birth Date), 77  
Indexes, Original table, 77  
Indexes, Table displayed according to Birth Date, 77  
Indexes, Table displayed according to Sex/Name tag, 78  
Inheritance, 194, 260  
Inheritance and overriding methods, 262  
Init, 231

INSERT, 143  
Installing and starting Visual FoxPro, 5  
Instance, 260  
Instances, 70  
Instantiation, 260  
Integrated Development Environment, 7  
Integrated toolbar, 373  
Integrating forms and data, 206  
Interactive Tab Ordering, 219  
International, 43  
Interrupts (Mouse Keyboard Other), 42  
ISAM, 69  
JOIN, 134  
Kevin McNeish, 476  
Key Label, 180  
Keyboard hotkeys, 237  
Keys, 71  
Label, 9  
Label Designer, 305  
Labels, Avery, 305  
Layout Controls toolbar, 301  
LEFT JOIN, 134  
Limits, 21  
List Box, 10  
LOCATE, 114  
Location parameter, 455  
Logical fields, 74  
LostFocus, 231  
Low-level file functions, 44  
Macros, 45  
Manipulating databases, 93  
Manipulating tables, 81  
Manipulating tables, Accessing an existing table, 82  
Manipulating tables, Accessing multiple tables, 84  
Manipulating tables, Closing a table, 86  
Manipulating tables, Creating indexes, 88  
Manipulating tables, Exclusive vs. Shared use, 83  
Manipulating tables, Specifying the order of records, 88  
Manipulating tables, Viewing the contents of, 87  
Map field types to your own classes, 361  
Math, 45  
Math - Financial, 46  
Math - Numeric, 46  
Math - Statistical, 47  
Math - Trig and Exponential, 47  
Memo binary, 75  
Memo fields, 74  
Memory, 47  
Menu, 8, 173  
Menu Building, 447  
Menu Designer, 176  
Menu Designer, Adding flexibility to, 187  
Menu Designer, Cleanup Code, 182  
Menu Designer, Creating context menus, 189  
Menu Designer, Dialog, 176  
Menu Designer, Menu menu, 180  
Menu Designer, Object Oriented, 191  
Menu Designer, Quick Menu, 180  
Menu Designer, Setup Code, 181  
Menu Designer, Top-Level Forms, 184  
Menu Designer, View menu, 181  
Menu Level, 178  
Menu pad, 8  
Menu styles, CUA, 314  
Menu styles, Document-centric, 315  
Menu styles, Function-centric, 313  
Menu styles, Pseudo-object-oriented, 315  
Menu-generation process, 184  
MessageBox(), 390  
Metadata, 458  
Metamor Worldwide, 482  
Method, 196  
Method parameter, 455  
Micromega Systems, 473  
Microsoft Knowledge Base, 595  
Microsoft Transaction Server, 703  
Microsoft Transaction Server, Creating a package, 705  
Microsoft Transaction Server, Installing and accessing, 704  
Microsoft Transaction Server, More information, 712  
Miscellaneous, 48  
MODIFY DATABASE, 94  
MODIFY STRUCTURE, 89, 99  
Modifying Data, 115  
Modifying tables and databases programmatically, 107  
Modifying the structure of an existing table, 91  
Movable, 253  
MoverBars property, 250  
MSDN, 595, 451  
MSDN Library, 581  
Multi-Table SELECT's, 130  
Multiple sets of data, 306  
MultiSelect property, 250, 251  
Navigating through tables and finding specific data, 109  
Negotiate, 180  
NODEFAULT, 284  
Non-Default Properties Only, 211  
Non-visual class, Creating, 329  
Non-visual class, Enhancing, 339  
Non-visual class, Implementing, 335  
Non-visual classes, 263  
NOT, 21  
Nulls, 75  
Numeric fields, 74  
Oak Leaf Enterprises, 476  
OApp.DataIsGood(), 344  
OApp.GetDiskSpace(), 344  
OApp.GetRAM(), 344

OApp.GetRegValue(), 344  
OApp.Init(), 343  
OApp.it(), 343  
OApp.login(), 343  
OApp.readini(), 343  
Object code, 200  
Object combo box, 210  
Object drop-down, 228  
Object orientation, 279  
Object orientation, Adding your own properties and methods, 280  
Object orientation, Naming classes, 279  
Object orientation, Overriding method inheritance, 283  
Object orientation, Overriding property inheritance, 282  
Object orientation, Preventing a parent's method from firing, 283  
Object orientation, Preventing an event from firing, 284  
Object orientation, Property and method hierarchy, 282  
Object orientation, Suggested base class modifications, 285  
Object-oriented programming, 259  
Object-oriented programming, Terminology, 261  
Obsolete commands and functions, 63  
OLE DB, 677  
OLE Drag and Drop, 17  
OLEDragDrop event, 516  
OLEDragOver event, 516  
OOP, 48  
OOP - Control-specific, 49  
Open Catalog, 497  
Operators, 20  
Option Group, 9  
Options, 179  
OR, 21  
ORDER BY, 141  
Other FoxPro files, 80  
Outer joins, 131  
Overriding the method code, 263  
Pad Name, 178  
Page Frame, 11  
Parent, 223  
Parent class, 262  
PEM's, 13  
Persistent relationships, 101  
Primary key, 71  
Printing, 50  
Procedure file, 430  
Procedure option, 177  
Procedure/Function, 155  
Processes menu, 318  
ProgID, 675  
Program control structures, 51  
Program event handling, 52  
Program subroutines, 52  
Program variable scoping, 53  
Program, Compiling, 152  
Program, Creating with the editor, 149  
Program, Running, 151  
Programming - Comments, 51  
Project Hook class, 505, 518  
Project Hooks, 514  
Project Hooks, Available functions that can be intercepted, 514  
Project Hooks, Enabling, 516  
Project Hooks, Enabling a global project hook, 518  
Project Hooks, Enabling for a specific project, 516  
Project Hooks, Examples, 520  
Project Hooks, Getting in and around a ProjectHook setting, 518  
Project Info, 167  
Project manager, 53  
Project Manager, 162  
Project Manager context menu, 165  
Project Manager, context menu, 165  
Project Manager, dialog, 162  
Project Menu, 169  
Project object, 505  
Project tools, 511  
Projects collection, 507  
Prompt column, 177  
PROMPT(), 186  
Properties, 195  
Properties events and methods, 13  
Property Settings, 212  
Query AddFile event, 515  
QueryModifyFile event, 515  
QueryRemoveFile event, 515  
QueryRunFile event, 515  
READ EVENTS, 323  
ReadOnly properties, 253  
RECALL, 145  
Record, 70  
Record is in use by another, 394  
Record subsets, 126  
RecordMark, 253  
Records, Adding interactively, 115  
Records, Adding programmatically, 115  
Records, Deleting interactively, 117  
Records, Deleting programmatically, 117  
Records, Editing interactively, 119  
Records, Editing programmatically, 119  
Records, Recalling and packing deleted records, 117  
Recursive, 496  
Refresh(), 421  
Regenerate Component, 165  
Registered classes, 354  
Relational model, 69, 70  
RemoveAppToolBar(), 386

REPLACE, 120  
Report Controls toolbar, 301  
Report Designer, 298  
Report Designer window, 298  
Report Designer window, Select a group of controls, 300  
Report Designer window, Delete a control, 300  
Report Designer window, Duplicate a control, 300  
Report Designer window, Move a control within a report, 300  
Report Designer window, Resize a control, 300  
Report Designer window, Select a control, 299  
Report menu, 302  
Report menu, Data Grouping menu option, 302  
Report menu, Group properties, 302  
Report menu, Reprint group header on each page, 302  
Report menu, Reset page number to 1 for each group check box, 302  
Report menu, Start each group on a new page check box, 302  
Report menu, Start group on new column check box, 302  
Report menu, Start group on new page when less than, 302  
Report menu, Title/Summary menu option, 302  
Report terminology, 293  
Report Writer, Dumbing down, 295  
Report, Detail band, 299  
Report, Detail row, 293  
Report, Footer, 294  
Report, Group, 294  
Report, Group footer, 294, 299  
Report, Group header, 294, 299  
Report, Header, 293  
Report, Page footer, 299  
Report, Page header, 299  
Report, Summary, 294  
Report, Summary band, 299  
Report, Title, 294  
Report, Title band, 299  
Reports menu, 318  
Resizable, 253  
Resolution, Ltd., 484  
Resource File, 447  
RestoreSets(), 339  
Restoring the toolbar, 432  
Result drop-down option, 177  
RGB(), 229  
Rick Strahl, 483  
RIGHT JOIN, 134  
Right-click menu, 8  
RowSourceType, 246  
Running a program, 151  
RUSHMORE, 79, 113  
Save(), 396  
SaveSets(), 339  
SCATTER, 120  
Scientific method, 554  
ScrollBars, 253  
Search Path, 448  
SEEK, 112  
SELECT, 123  
SELECT, Basic syntax, 124  
SELECT, Multi-Table, 130  
Set as Default, 153  
SET ASSERTS ON, 449  
SET DATABASE TO, 94  
SET DEBUGOUT TO, 525  
SET DEFAULT TO, 150  
SET DEVELOPMENT, 159  
SET ESCAPE OFF, 159  
SET SYSMENU NOSAVE, 186  
SET SYSMENU SAVE, 449  
SET SYSMENU TO DEFAULT, 449  
Setting a filter, 92  
Shortcut Designer, 189  
Shortcut menu, 8  
ShowAppToolBar(), 386  
Single key, 71  
SKIP, 109  
Skip For, 180  
Sort Order combo box, 411  
Source code, 457  
Spinner, 11  
Start in property, 444  
Startup configuration, 443  
Startup files, 445  
Startup files, CONFIG.FPW, 446  
Startup files, DEFAULT.FKY, 447  
Startup files, FoxPro.INI, 447  
Startup files, FoxUser.DBF, 446  
Startup files, The Windows Registry, 445  
Startup Program, 448  
Startup switches, 452  
Steven Black, 475  
Stonefield Systems, 479  
String functions, 54  
Submenu option, 178  
Subtotaling - GROUP BY, 130  
Superclass, 262  
Superkey, 71  
SYS(2003), 150  
SYS(5), 150  
Tabbed dialog, 11  
Table structures, working with programmatically, 102  
Tables, 70, 72  
Tables, Manipulating, 81  
Take Note Consulting, 473  
Tastrade, 312  
Text Box, 9  
Text merge, 56  
Third-party libraries and tools, 457

THISFORM, 196  
Timer's functionality, Caveats, 239  
Today's computing environment, 4  
Toolbar, 8  
Tools menu, 318  
Top-Level Forms, 184  
Trace window, Pause between line execution, 526  
Trace window, Show line numbers, 526  
Trace window, Trace between break points, 526  
Trace window, Trace option button, 526  
Tuple, 70  
UPDATE, 145  
User interface, 56  
Using the Menu Designer to create a menu, 174  
Using the VFP Project Manager to build an .APP or .EXE, 161  
Variables, 18  
Variables, 57  
VFP, 58  
VFP Editor, 152  
VFP's default directory, 444  
View menu, 302  
View menu, Grid lines visible, 304  
View menu, Landscape report, 304  
View menu, Preview toolbar, 304  
View menu, Show Position menu option, 304  
View Type combo, 488  
Viewing the contents of a database, 94  
Viewing the contents of a database, 94  
Viewing the contents of a table, 87  
Visual Basic, 713  
Visual Basic, Adding code to a form, 716  
Visual Basic, ADO in VB, 757  
Visual Basic, Arrays, 735  
Visual Basic, Booleans (logicals), 733  
Visual Basic, Classes, 743  
Visual Basic, Commands, 738  
Visual Basic, Constants, 734  
Visual Basic, Data Conversion, 736  
Visual Basic, Data manipulation, 736  
Visual Basic, Date and time, 737  
Visual Basic, Dates, 733  
Visual Basic, File handling, 741  
Visual Basic, File system, 742  
Visual Basic, Functions, 735  
Visual Basic, Logic structures, 738  
Visual Basic, Math functions, 738  
Visual Basic, Number handling, 736  
Visual Basic, Operators, 731  
Visual Basic, Program control, 742  
Visual Basic, Project components and files, 719  
Visual Basic, Registry, 743  
Visual Basic, Saving and printing projects, 719  
Visual Basic, Special functions, 738  
Visual Basic, Static variables, 735  
Visual Basic, String manipulation, 738  
Visual Basic, The basics of writing code, 722  
Visual Basic, The Immediate window, 730  
Visual Basic, The VB code window, 718  
Visual Basic, The Visual Basic IDE, 714  
Visual Basic, Times, 733  
Visual Basic, Variables, 730  
Visual Basic, Variables, 742  
Visual Basic, Variants, 733  
Visual Basic, Visual Basic classes?, 771  
Visual delineators, 234  
Visual delineators, Image control, 234  
Visual delineators, Label control, 234  
Visual delineators, Line control, 234  
Visual delineators, Page frame control, 234  
Visual delineators, Separator control, 234  
Visual delineators, Shape control, 234  
Visual Fox Pro, interface, 5  
Visual FoxPro technology, 558  
Visual FoxPro's data structures, 72  
Visual FoxPro, command and function overview, 22  
Visual FoxPro, Configuring program execution preferences, 159  
Visual FoxPro, default directory, 444  
Visual FoxPro, Editor, 152  
Visual FoxPro, Installing and Starting, 5  
Visual FoxPro, Native Controls, 227  
Visual FoxPro, Other files, 80  
Visual FoxPro, Setup Wizard, 647  
Visual InterDev, 673  
Visual property, 237  
Visual Studios, 59  
Wayne Ratliff, 3, 72  
West Wind Technologies, 475  
WHERE, 126  
Which band is printed when?, 298  
Window menu, 319  
Windows API, 613  
Windows API, Pointers as return values, 619  
Windows API, Structures, 618  
Windows API, What do functions look like in the Windows API?, 614  
Windows menus and toolbars, 8  
Windows Quick Launch toolbar, 444  
Windows/menus, 60  
Working with data, 109  
Working with data structures, 81  
\_INCLUDE, 220  
\_screen, 223