

visual basic® 2005

DeMYSTiFieD

A SELF-TEACHING GUIDE



LEARN to build PROGRAMS for
WINDOWS and the WEB



Get up to speed FAST



Covers BOTH the LANGUAGE and
its LIBRARIES



Complete with chapter-ending
QUIZZES and final EXAM

Mc
Graw
Hill

Jeff Kent





VISUAL BASIC 2005 DEMYSTIFIED

This page intentionally left blank



VISUAL BASIC 2005 DEMYSTIFIED

JEFF KENT

McGraw-Hill

New York Chicago San Francisco Lisbon London
Madrid Mexico City Milan New Delhi San Juan
Seoul Singapore Sydney Toronto

Copyright © 2006 by The McGraw-Hill Companies. All rights reserved. Manufactured in the United States of America. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

0-07-148675-5.

The material in this eBook also appears in the print version of this title: 0-07-226171-4.

All trademarks are trademarks of their respective owners. Rather than put a trademark symbol after every occurrence of a trademarked name, we use names in an editorial fashion only, and to the benefit of the trademark owner, with no intention of infringement of the trademark. Where such designations appear in this book, they have been printed with initial caps.

McGraw-Hill eBooks are available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. For more information, please contact George Hoare, Special Sales, at george_hoare@mcgraw-hill.com or (212) 904-4069.

TERMS OF USE

This is a copyrighted work and The McGraw-Hill Companies, Inc. (“McGraw-Hill”) and its licensors reserve all rights in and to the work. Use of this work is subject to these terms. Except as permitted under the Copyright Act of 1976 and the right to store and retrieve one copy of the work, you may not decompile, disassemble, reverse engineer, reproduce, modify, create derivative works based upon, transmit, distribute, disseminate, sell, publish or sublicense the work or any part of it without McGraw-Hill’s prior consent. You may use the work for your own noncommercial and personal use; any other use of the work is strictly prohibited. Your right to use the work may be terminated if you fail to comply with these terms.

THE WORK IS PROVIDED “AS IS.” MCGRAW-HILL AND ITS LICENSORS MAKE NO GUARANTEES OR WARRANTIES AS TO THE ACCURACY, ADEQUACY OR COMPLETENESS OF OR RESULTS TO BE OBTAINED FROM USING THE WORK, INCLUDING ANY INFORMATION THAT CAN BE ACCESSED THROUGH THE WORK VIA HYPERLINK OR OTHERWISE, AND EXPRESSLY DISCLAIM ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. McGraw-Hill and its licensors do not warrant or guarantee that the functions contained in the work will meet your requirements or that its operation will be uninterrupted or error free. Neither McGraw-Hill nor its licensors shall be liable to you or anyone else for any inaccuracy, error or omission, regardless of cause, in the work or for any damages resulting therefrom. McGraw-Hill has no responsibility for the content of any information accessed through the work. Under no circumstances shall McGraw-Hill and/or its licensors be liable for any indirect, incidental, special, punitive, consequential or similar damages that result from the use of or inability to use the work, even if any of them has been advised of the possibility of such damages. This limitation of liability shall apply to any claim or cause whatsoever whether such claim or cause arises in contract, tort or otherwise.

DOI: 10.1036/0072261714



Professional



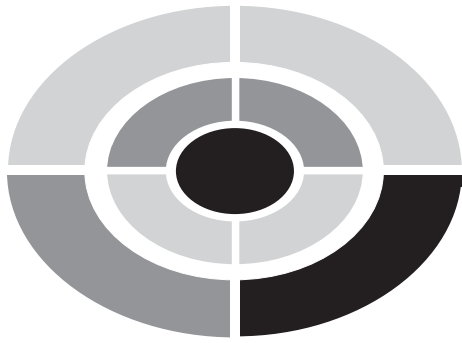
Want to learn more?

We hope you enjoy this
McGraw-Hill eBook! If

you'd like more information about this book,
its author, or related books and websites,
please [click here](#).

I would like to dedicate this book to my younger daughter, Emily Rebecca Kent. Being a teenager is never easy and even more difficult during these times. Emily, I am proud of your recent progress, and hope it continues as you approach, all too soon for a parent, becoming a young adult. I love you very much.

—Dad



ABOUT THE AUTHOR

Jeff Kent is an Associate Professor of Computer Science at Los Angeles Valley College in Valley Glen, California. He teaches a number of programming languages, including Visual Basic, C++, C#, Java, and, when he's feeling masochistic, Assembler. He also manages a network for a Los Angeles law firm whose employees are guinea pigs for his applications, and as an attorney gives advice to young attorneys whether they want to hear it or not. He also has written several books on computer programming, recently *Visual Basic.NET: A Beginner's Guide* and *C++ Demystified* (McGraw-Hill/Osborne), and, concurrently with this book, *Visual C# 2005 Demystified* (McGraw-Hill).

Jeff has had a varied career—or careers. He graduated from UCLA with a Bachelor of Science degree in economics and then went on to obtain a Juris Doctor degree from Loyola (Los Angeles) School of Law and to practice law. During this time, when personal computers were still a gleam in Bill Gates's eye, Jeff was also a professional chess master, earning a third place finish in the United States Under-21 Championship and, later, an international title.

Jeff does find time to spend with his wife, Devvie, which is not difficult since she is also a computer science professor at Valley College. In addition to his other career pursuits, he has a part-time job as personal chauffeur for his teenage daughter Emily (his older daughter Elise now has her own driver's license), and in what little spare time he has, he enjoys watching international chess tournaments on the Internet. His goal is to resume running marathons, since otherwise, given his losing battle to lose weight, his next book may be *Sumo Wrestling Demystified*.

CONTENTS AT A GLANCE



PART ONE	INTRODUCTION TO VISUAL BASIC 2005	
CHAPTER 1	Getting Started with Your First Windows Program	3
CHAPTER 2	Writing Your First Code	19
CHAPTER 3	Controls	43
PART TWO	PROGRAMMING BUILDING BLOCKS: VARIABLES, DATA TYPES, AND OPERATORS	
CHAPTER 4	Storing Information—Data Types and Variables	65
CHAPTER 5	Letting the Program Do the Math—Arithmetic Operators	79
CHAPTER 6	Making Comparisons—Comparison and Logical Operators	97
PART THREE	CONTROLLING THE FLOW OF THE PROGRAM	
CHAPTER 7	Making Choices—If and Select Case Control Structures	115
CHAPTER 8	Repeating Yourself—Loops and Arrays	139
CHAPTER 9	Organizing Your Code with Procedures	157



PART FOUR	THE USER INTERFACE	
CHAPTER 10	Helper Forms	179
CHAPTER 11	Menus	197
CHAPTER 12	Toolbars	221
PART FIVE	ACCESSING DATA	
CHAPTER 13	Accessing Text Files	239
CHAPTER 14	Databases	255
CHAPTER 15	Web Applications	277
	Final Exam	301
	Answers to Quizzes and Final Exam	307
	Index	325

CONTENTS



Acknowledgments	xix
Introduction	xxi

PART ONE

INTRODUCTION TO VISUAL BASIC 2005

CHAPTER 1

Getting Started with Your First Windows Program	3
Obtaining and Installing Visual Basic 2005	4
System Requirements	5
Choosing the Right Version	6
Installing Visual Basic 2005	6
Starting Your First Visual Basic 2005 Project	7
Starting the Program	7
Specifying the Type of New Project	7
Specifying the Name and Location of the Project	10
Integrated Development Environment (IDE)	11
Run the Project!	12
What Is a Computer Program?	14
What Is a Programming Language?	15
Translating the Code for the Computer	16
Conclusion	17
Quiz	17





CHAPTER 2	Writing Your First Code	19
	Starting an Existing Project	20
	Design View and Code View	22
	Object Browser	25
	Classes and Objects	26
	Inherits	27
	Namespaces	27
	.NET Framework	28
	Properties	28
	Properties Window	29
	Changing Properties at Design Time	31
	What Is a Windows Application?	31
	Windows Applications Are Gooney	32
	Windows Applications Are Event-Driven	34
	Classes Have Events	35
	Creating an Event Procedure	35
	Creating an Event Procedure Stub	36
	Writing Code Inside the Event Procedure	38
	Assignment Operator	39
	Comments	40
	Conclusion	41
	Quiz	41
CHAPTER 3	Controls	43
	Adding Controls to the Form	44
	Toolbox	44
	Copying a Control from the Toolbox to the Form	46
	Changing the Control's Location	46
	Changing the Control's Size	48
	Important Label Properties	50
	Text Property	50
	Name Property	51



The Label Control in Action	52
Mouse Coordinates	52
Creating the Application	53
How the Code Works	56
Line-Continuation Character	56
Using Event Procedure Parameters	56
Handles Clause	57
What If You Type the Wrong Code?	57
Conclusion	59
Quiz	60

PART TWO PROGRAMMING BUILDING BLOCKS: VARIABLES, DATA TYPES, AND OPERATORS

CHAPTER 4	Storing Information—Data Types and Variables	65
	Data Types	66
	Numeric Data Types	67
	Text Data Types	68
	Data Types of Visual Basic Properties	68
	Variables	70
	Declaring a Variable	70
	Where Do You Declare a Variable?	72
	Constants	75
	Declaring a Constant	75
	Why Use Constants?	76
	Conclusion	77
	Quiz	77
CHAPTER 5	Letting the Program Do the Math—Arithmetic Operators	79
	Arithmetic Operators	80
	The Addition Operator	80
	The Subtraction Operator	81
	The Multiplication Operator	81



The Exponent Operator	81
The Division Operators	82
Operator Precedence	83
Combining Arithmetic and Assignment Operators	83
The Parse and ToString Methods	84
Class Methods	86
Change Machine Project	86
Creating the Project	87
The Algorithm	90
Type Conversions	92
Conclusion	94
Quiz	95

CHAPTER 6

Making Comparisons—Comparison and Logical Operators	97
Debugging	98
Comparison Operators	100
Numeric Comparison Operators	100
String Comparisons	102
Precedence	105
Logical Operators	106
And Operator	106
AndAlso Operator	107
Or Operator	108
OrElse Operator	109
Xor Operator	109
Not Operator	110
Precedence	110
Why AndAlso and OrElse in Addition to And and Or?	111
Conclusion	111
Quiz	112



PART THREE	CONTROLLING THE FLOW OF THE PROGRAM	
CHAPTER 7	Making Choices—If and Select Case Control Structures	115
	The InputBox Function	116
	Modal vs. Modeless	116
	Displaying an Input Box	118
	Return Value	118
	If Control Structure	119
	If...Then Statement	120
	If...Then...Else Statement	121
	If...ElseIf Statement	122
	Input Validation	124
	Exceptions	124
	Controls Used for If Control Structure	128
	CheckBox Control	128
	RadioButton Control	130
	Pizza Calculator	131
	Creating the Project	131
	How the Project Works	132
	The Code	132
	Select Case Control Structure	134
	Syntax	135
	The Is Keyword	135
	Select Case Control Structure in Action	136
	Choosing Between If...ElseIf and Select Case	137
	Conclusion	137
	Quiz	138
CHAPTER 8	Repeating Yourself—Loops and Arrays	139
	Loops	140
	For...Next Statement	140



	While...End While Statement	147
	Do Statement	150
	For Each...Next Loop	152
	Arrays	152
	Declaring Arrays	153
	Default Value	153
	Conclusion	155
	Quiz	155
CHAPTER 9	Organizing Your Code with Procedures	157
	Types of Procedures	158
	Built-In vs. Programmer-Defined Procedures	158
	Methods Contrasted	159
	Subroutines	159
	Declaring a Subroutine	159
	Calling the Subroutine	162
	Parameters	163
	Functions	168
	Declaring Functions	168
	Calling Functions	170
	How the Value Is Returned	171
	Why Write Your Own Procedures?	174
	Conclusion	175
	Quiz	176
PART FOUR	THE USER INTERFACE	
CHAPTER 10	Helper Forms	179
	Message Boxes	180
	Creating the Project	181
	Message Boxes Are Modal	182

	Show Method	182
	Using the Show Method's Return Value	186
	Dialog Forms	188
	Creating the Project	188
	Showing the Dialog Form and Returning Its Result	192
	Accessing Values from the Dialog Form	194
	Modal vs. Modeless	194
	Conclusion	195
	Quiz	195
CHAPTER 11	Menus	197
	Creating a Main Menu	198
	Adding a MenuStrip Control to a Form	199
	Adding Menu Items to the MenuStrip	200
	Enhancing the Menu Items	204
	Adding Functionality to the Menu Items	207
	Disabling Menu Items	208
	Creating a Context Menu	209
	Adding a ContextMenuStrip to a Form	210
	Adding Menu Items to the ContextMenuStrip	211
	Adding Functionality to Context Menu Items	214
	Text Editor Project	217
	Creating the Project	217
	Explanation of the Code	219
	Conclusion	220
	Quiz	220
CHAPTER 12	Toolbars	221
	Creating a Toolbar	222
	Adding a Toolbar to a Form	222



Adding Buttons to the Toolbar	224
Associating Images with Toolbar Buttons	227
Associating Code with Clicks of Toolbar Buttons	233
Conclusion	234
Quiz	234

PART FIVE ACCESSING DATA

CHAPTER 13 Accessing Text Files 239

Open and Save File Dialog Boxes	240
Adding an OpenFileDialog Control to Your Form	240
Showing the OpenFileDialog Control	241
Determining Whether Open or Cancel Has Been Chosen	242
Identifying the File to Open	243
SaveFileDialog Control	244
Reading from a Text File	246
StreamReader Class	246
Reading the Text File into the TextBox	248
Closing the Text File	249
Writing to a Text File	250
StreamWriter Class	250
Writing from the TextBox to the Text File	251
Closing the Text File	252
Conclusion	253
Quiz	253

CHAPTER 14 Databases 255

Installing the Database	256
Obtaining the Northwind Traders Database	256
Installing the Northwind Traders Database	256
Connecting to the Database	257



Using Server Explorer	260
Exploring the Database	261
Exploring the Customers Table	262
Database Project	264
What the Project Does	264
Creating the Form	264
Importing Data Namespaces	266
Creating a Connection	267
Creating a Command	269
Filling the DataGridView	271
Conclusion	275
Quiz	276
CHAPTER 15	
Web Applications	277
ASP.NET	278
Internet Information Services	278
Determining If IIS Is Already Installed	279
Installing IIS	280
Start the IIS Admin Service	280
Starting the Default Website	282
URL	284
Your Computer as the Web Server	284
Virtual and Physical Paths	285
Creating a Web Application	287
ASP.NET Development Server	288
ASP.NET Application IDE	290
Creating a Database Web Application	292
Adding a GridView Control	292
Locating the Database on the Web Server	295
Adding Code	297
Conclusion	298
Quiz	298



Final Exam	301
Answers to Quizzes and Final Exam	307
Index	325

ACKNOWLEDGMENTS



It seems obligatory in acknowledgments for authors to thank their publishers (especially if they want to write for them again), but I really mean it. This is my fifth book for McGraw-Hill, and I hope there will be many more. It truly is a pleasure to work with professionals who are nice people as well as very good at what they do (even when what they are very good at is keeping accurate track of the deadlines I miss).

I first want to thank Wendy Rinaldi, who got me started with McGraw-Hill/Osborne back in 1998 (has it been that long?). Wendy was also my first acquisitions editor. She has since received several well-deserved promotions, but is still my acquisitions editor. Indeed, this book was launched through a telephone call with Wendy at the end of a vacation with my wife, Devvie, who, being in earshot and with an are-you-insane tone in her voice, asked incredulously, “You’re writing another book?”

I replied, “Of course not, honey...”

She interjected, “That’s a relief!”

I then continued, “...I’m writing two books.” (I wrote *Visual C# 2005 Demystified* concurrently with this book).

I must also thank my acquisitions coordinator, Alexander McDonald, and my project editor, LeeAnn Pickrell. Both were unfailingly helpful and patient, while still keeping me on track in this deadline sensitive business (e.g., I’m so sorry you broke both your arms and legs; you’ll still have the next chapter turned in by this Friday, right?”).

Bart Reed did the copyediting. He was kind about my obvious failure during my school days to pay attention to my grammar lessons. He improved what I wrote while still keeping it in my words (that way if something is wrong it is still my fault).

Ron Petrusha was my technical editor. Ron’s suggestions were quite helpful and added a lot of value to this book.





There were many other talented people working behind the scenes who also helped get this book out to press, and as in an Academy Award speech, I can't list them all. That doesn't mean I don't appreciate all their hard work, because I do.

I truly thank my wife, Devvie, who in addition to being my wife, best friend (maybe my only one), and partner (I'm leaving out lover because computer programmers aren't supposed to be interested in such things), tolerated my incessant muttering about unreasonable chapter deadlines and merciless editors (sorry, Alex) while excusing myself from what she wanted to do (or wanted me to do). Similarly, I would like to give thanks to my daughters, Elise and Emily, and my mom, Bea Kent, for tolerating my absentmindedness while I was preoccupied with unreasonable chapter deadlines and merciless editors (starting to notice a pattern here?). I also should thank my family in advance for not having me committed when I talk about writing my next book.



INTRODUCTION

One of my favorite movie lines was in *Rocky III*: Before their rematch, Mr. T, playing a boxer called Clubber Lang who had beaten up Rocky badly in their first fight, said, “Fool, you never should have come back.”

Visual Basic must be saying this to me. A few years ago I wrote a book, *Visual Basic .NET: A Beginner’s Guide*, timed to be on the bookshelves for the release of Visual Basic .NET. Writing such a “day and date” book is added pressure, especially since Microsoft is famous (or infamous) for last minute changes from their most recent beta.

I must have a short memory or be a slow learner. With the next major change in Visual Basic, Visual Basic 2005, here I go again writing another “day and date” book.

Why Did I Write This Book?

Given my griping about writing another “day and date” book, you may be wondering why I wrote this book. I assure you that the reason was not because I thought it would get me riches, fame, or beautiful women. I may be misguided, but I’m not completely delusional or, in the case of my wife’s reaction to the beautiful women part, suicidal.

To be sure, there likely will be many introductory-level books on Visual Basic 2005. Nevertheless, I wrote this book because I believe I bring a different and, I hope, valuable perspective.

As you may know from my author biography, I teach computer science at Los Angeles Valley College, a community college in the San Fernando Valley area of Los Angeles, where I grew up and have lived most of my life. I also write computer programs, but teaching programming has provided me with insights into how students learn that I could never obtain from just writing programs. These insights are gained not just from answering student questions during lectures; I spend hours each week in our college’s computer lab helping students with their programs and



more hours each week reviewing and grading their assignments. Patterns emerge regarding which teaching methods work and which don't, the order in which to introduce programming topics, the level of difficulty at which to introduce a new topic, and so on. I joke with my students that they are my beta testers in my never-ending attempt to become a better teacher, but there is much truth in that joke.

Additionally, my beta testers... err, students, seem to complain about the textbook no matter which book I adopt. Many ask me why I don't write a book they could use to learn Visual Basic. They may be saying this to flatter me (I'm not saying it doesn't work), or for the more sinister reason that they will then be able to blame the teacher for a poor book as well as poor instruction. Nevertheless, having written other books, these questions planted in my mind the idea of writing a book that, in addition to being sold to the general public, could also be used as a supplement to a textbook.

Who Should Read This Book

Anyone who will pay for it! Just kidding, though no buyers will be turned away.

It is hardly news that publishers and authors want the largest possible audience for their books. Therefore, this section of the introduction usually tells you this book is for you—whoever you may be and whatever you do. However, no programming book is for everyone. For example, if you exclusively create game programs using Java, this book may not be for you (though being a community college teacher I may be your next customer if you create a space beasts vs. community college administrators game).

While this book is not for everyone, it may very well be for you. Many people need or want to learn Visual Basic, either as part of a degree program, job training, or even as a hobby. Unfortunately many books don't make learning Visual Basic any easier, throwing at you a veritable telephone book of complexity and jargon. By contrast, this book, as its title suggests, is designed to “demystify” Visual Basic. Therefore, it goes straight to the core concepts and explains them in logical order and in plain English.

What This Book Covers

I strongly believe that the best way to learn programming is to write programs. The concepts covered by the chapters are illustrated by programs you can write using tested and thoroughly explained code. You can run this code yourself, and also use it as the basis for writing further programs that expand on the covered concepts.



The first part of this book is designed to get you up and running with Visual Basic 2005. Chapter 1 is titled “Getting Started with Your First Windows Program.” The first step in programming in Visual Basic 2005 is to obtain and install it. This chapter advises you on how to do that. The chapter then shows you how to create your first Visual Basic 2005 project and concludes by defining core concepts such as computer program, programming language, and how your code is translated for the computer.

Chapter 1 teaches you how to create a working Windows application without having to write any code. However, you will need to write code for even the simplest program. Thus, Chapter 2 is about “Writing Your First Code.” This chapter explains key programming concepts, such as classes, objects, and properties, as well as giving you a tour of the Visual Basic 2005 Integrated Development Environment (IDE). The chapter then describes the event-driven nature of a Windows application and shows you how to put this theory into practice by creating an event procedure.

Chapters 1 and 2 focus on the form, perhaps the most important part of a Windows application’s graphical user interface, or GUI. However, a form cannot possibly meet all the requirements of a Windows application. For example, the form does not have the functionality to permit the typing of text, listing of data, selecting of choices, and so forth. You need other, specialized controls for that additional functionality. Indeed, the form’s primary role is to serve as a host, or container, for other controls that enrich the GUI of Windows applications, such as menus, toolbars, buttons, text boxes, and list boxes. Chapter 3, “Controls,” explains how to add controls to your form and manipulate their properties. This chapter then uses a project to demonstrate how you can use a control’s events in an application.

Now that you are up and running with Visual Basic 2005, the next part of this book covers the building blocks of your programs: variables, data types, and operators, starting with Chapter 4, “Storing Information—Data Types and Variables.” Most computer programs store information, or data. Data comes in different varieties, such as numeric or text. The type of information, whether numeric, text, or Boolean, is referred to as the *data type* and often is stored in a variable, which not only reserves the amount of memory necessary to store information, but also provides you with a name by which that information may be retrieved later. Finally, this chapter covers constants, which are similar to variables, but differ in that their initial value never changes while the program is running.

As a former professional chess player, I have marveled at the ability of chess computers to play world champions on even terms. The reason the chess computers have this ability is because they can calculate far more quickly and accurately than we can. Chapter 5, “Letting the Program Do the Math—Arithmetic Operators,” covers arithmetic operators, which we use in code to harness the computer’s calculating ability.

Now that we have covered the programming building blocks, we'll use them in the next part of this book to control the flow of your program. As programs become more sophisticated, they often branch in two or more directions based on whether a condition is true or false. For example, while a calculator program would use the arithmetic operators you learned about in Chapter 5, your program first needs to determine whether the user chose addition, subtraction, multiplication, or division before performing the indicated arithmetic operation. Chapter 6, "Making Comparisons—Comparison and Logical Operators," introduces comparison and logical operators, which are useful in determining a user's choice. Chapter 7, "Making Choices—If and Select Case Control Structures," introduces the If and Select Case statements, which are used to direct the path the code will follow based on the user's choice.

When you were a child, your parents may have told you not to repeat yourself. However, sometimes your code needs to repeat itself. For example, if an application user enters invalid data, your code may continue to ask the user whether they want to retry or quit until the user either enters valid data or quits. Chapter 8, "Repeating Yourself—Loops and Arrays" introduces loops, which are used to repeat code execution until a condition is no longer true. This chapter then discusses arrays. Unlike the variables we have covered thus far in the book, which may hold only one value at a time, arrays may hold multiple values at one time. Additionally, arrays work very well with loops.

This book is several hundred pages long. Imagine how much harder this book would be to understand if it consisted of only one, very long chapter, rather than being divided into chapters, with each chapter being divided into sections? Chapter 9, "Organizing Your Code with Procedures," shows you how to similarly divide up your code into separate procedures. This has advantages in addition to making your code easier to understand. For example, if a method performs a specific task such as sending output to a printer, which is performed several times in a program, you only need to write the code necessary to send output to the printer once, and then you call that method each time you need to perform that task. Otherwise, the code necessary to send output to the printer would have to be repeated each time that task had to be performed. Further, if you later have to fix a bug in how you perform that task, or simply find a better way to perform the task, you have to change the code in only one place rather than many.

The next part of this book focuses on the graphical user interface (GUI), starting with Chapter 10, "Helper Forms." Up until now, our applications have had one form that serves as the main application window. This one form may be sufficient for a simple application, but as your applications become more sophisticated, the main application form will become unable to perform all the tasks required by the application and need help from other forms. This chapter shows you how to create and use two helper forms that will be workhorses in your applications: message boxes and dialog forms. While these helper forms are, well, helpful, they also



present programming challenges involving communication between the main and helper form. For example, the main form needs to know which button was clicked on the helper form so it can execute the appropriate code depending on which button was clicked. Additionally, since the dialog form contains controls, the main form needs to know and take actions based on what the application user types, checks, or selects in the controls in the helper form. This chapter will show you how to solve these programming challenges.

Through the GUI of the application, users issue commands to an application, such as to open, save, or close a file, print a document, and so on. Chapter 11, “Menus,” and Chapter 12, “Toolbars,” cover the three most common GUI elements through which users give commands to an application: the menu, shortcut or context menus, and toolbars. Additionally, commands such as cut, copy, and paste are often duplicated in a menu, a context menu, and a toolbar, providing the user with the convenience of having three different ways to perform the same command. However, you don’t want to write the same code three times, so these chapters show you how to connect corresponding items in menus, context menus, and toolbars so they each execute the same code.

When I was finished writing this book for the evening, I closed Microsoft Word, and maybe even shut down my computer. Of course, the next evening I did not have to start over; what I had written the previous evening had been saved. However, up until now the programs in this book don’t save data so it is available even after the application exits. The next part of this book shows you how to save data. Chapter 13, “Accessing Text Files,” explains how to write code that reads from and writes to a text file. This chapter also shows you how to add Open and Save dialog boxes, such as those used in sophisticated programs like Microsoft Word, so you can open a text file to read from it and save to a text file to write to it. Chapter 14, “Databases,” explains how to write programs that access information stored in a database.

Throughout this book we have been writing Windows applications, which to be sure are used heavily. However, many of us interact ever more frequently with the subject of Chapter 15, “Web Applications.” This chapter shows you how to create a Web application that displays information from a database, similar to the Windows application you’ll create in Chapter 14.

How to Read This Book

I have organized this book to be read from beginning to end. While this may seem patently obvious, my students often express legitimate frustration about books (or teachers) that, in discussing a programming concept, mention other concepts that are covered several chapters later or, even worse, not at all. Therefore, I have endeavored to present the material in a linear, logical progression. This not only avoids the frustration of material that is out of order, but also enables you in each succeeding chapter to build on the skills you learned in the preceding chapters.



Special Features

Each chapter has detailed code listings so you can put into practice what you learned. My overall objective is to get you up to speed quickly, without a lot of dry theory or unnecessary detail. So let's get started. It's easy and fun to write Visual Basic programs.

Contacting the Author

Hmmm... it depends why. Just kidding. While I always welcome gushing praise and shameless flattery, comments, suggestions, and yes, even criticism can also be valuable. The best way to contact me is via email; you can use jkent@genhiskhent.com (the domain name is based on my students' fond (?) nickname for me, Genghis Khent). Alternately, you can visit my website, <http://www.genhiskhent.com/>. Don't be thrown off by the entry page; I use this site primarily to support the online classes and online components of other classes that I teach at the college, but there will be a link to the section that supports this book.

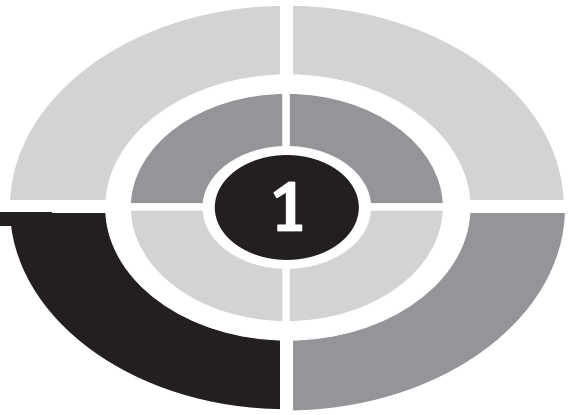
I hope you enjoy reading this book as much as I enjoyed writing it.



PART ONE

Introduction to Visual Basic 2005

This page intentionally left blank



CHAPTER

Getting Started with Your First Windows Program

You probably have seen on television an interviewer ask a victorious athlete for the secret of their success. Can you imagine the athlete replying that they never trained but instead just read about their sport a lot? I doubt it. The only way to become a good swimmer, runner, or weightlifter is to swim, run, or lift weights. Of course, good coaching helps, but a good swimmer must swim, a good runner must run, and a good weightlifter must lift weights.



Although computer programming is mental rather than physical exercise, similarly you cannot become a good computer programmer only by reading about computer programming. Instead, you have to write computer programs—lots of them.

Don't get me wrong, I'm not trying you to discourage you from buying a book, especially this one! A good book is like a good coach, making your learning more efficient and less frustrating. However, even with the best book, if you don't write computer programs, it will be difficult for you to learn computer programming. Fortunately, it is easy to start writing computer programs; this chapter will show you how.

Newcomers to programming sometimes shy away from writing programs because something may go wrong in their program. They may think of a scene in an action movie where someone has only seconds to defuse a bomb and they have to guess which one of several wires to cut. The consequences in those circumstances of making a mistake are life and death.

However, you are not defusing a bomb. You are writing a computer program. If you do make a mistake in your program, neither you nor your computer will disappear in a fireball. You just correct the mistake. Indeed, you learn best from your mistakes.

Since I have given you this speech on the importance of writing programs, it is only fair that I help you get started writing programs. The first step is for you to obtain and install Visual Basic 2005. In this chapter, I first will help you choose the edition of Visual Basic 2005 that is best for you, and assist you in ensuring that your computer meets the hardware requirements of Visual Basic 2005. After you install Visual Basic 2005, I will show you how to use it to create a Windows application. Finally, you will learn just what a computer program is.

Obtaining and Installing Visual Basic 2005

Visual Basic 2005 comes in several editions. This section will help you choose the one right for you. However, before you buy any edition of Visual Basic 2005, you should confirm that the computer on which you will install Visual Basic 2005 meets the hardware requirements of Visual Basic 2005.

Once you have purchased Visual Basic 2005 and verified that the installation computer meets the hardware requirements, you are ready to install Visual Basic 2005. This section will give you tips on the installation.

System Requirements

Installing Visual Basic 2005 requires not only the right software, but hardware sufficient to run the software. Therefore, you should first confirm that the computer on which you are going to install Visual Basic 2005 meets the system requirements, such as the operating system, processor, RAM, and available hard disk space.

NOTE *I will be referring in this chapter to Visual Basic 2005, but my comments apply whether you are buying Visual Basic 2005 alone or one of the editions of Visual Studio 2005, as discussed in the next section, “Choosing the Right Version.”*

Here are my recommendations on the key requirements. Keep in mind these system requirements are truly minimum, and therefore Visual Basic 2005 may run quite slowly if your computer only meets the bare minimum requirements.

- **Operating system** You must have Windows 2003, XP, or 2000; Windows NT, 95, 98, or Me will not work. If you have not yet purchased an operating system and are considering XP, I would recommend the Professional over the Home Edition, especially if you are developing web applications, which are discussed in Chapter 15.
- **Available hard drive space** The requirement varies with the edition and type of installation and whether other components such as Internet Explorer (IE) already are installed on your computer. You should plan on the total installation taking between 2GB and 5GB (gigabytes). A large (at least 80GB) hard drive is relatively inexpensive and easy to install, so if remaining space on your existing hard drive is scarce, you may wish to consider upgrading before installing Visual Basic 2005.
- **Processor** According to Microsoft, a processor speed of 600 MHz (megahertz) is the minimum and 1 GHz (gigahertz) is recommended. Because upgrading a processor by replacing the motherboard is not so inexpensive or easy, another alternative is boosting your system RAM, discussed next, if you are on the borderline.
- **RAM** According to Microsoft, 128MB (megabytes) is the minimum, and 256MB is recommended. I would recommend 512MB, especially if you are running other programs at the same time.

Additionally, Visual Basic 2005, in order to work properly, needs other software to be on your computer—in particular, Internet Explorer. If you are installing Visual Basic 2005 at work and your company restricts browsers to Netscape or another non-IE browser, you should check first with your system administrator before attempting to install Visual Basic 2005 there.

Choosing the Right Version

You can buy Visual Basic 2005 either by itself or as part of Visual Studio 2005, which includes, in addition to Visual Basic, support for other programming languages such as C++ and C#. I recommend Visual Studio 2005 if your budget allows. The additional cost usually is not that substantial and you will have a program that works with other commonly used languages if your education, employment, or interests prompt you to work with other programming languages. This is more likely than you may think. Once you learn one programming language, learning additional ones becomes much easier because the concepts are essentially the same. Indeed, most programmers don't learn just one language.

If you buy Visual Basic 2005 by itself, you have one choice: the Express Edition. If you instead buy Visual Basic 2005 as part of Visual Studio 2005, you have three choices: Standard, Professional, and Team System Editions.

If you already have a copy of Visual Basic 2005 through your school or job, any of these choices should work fine for this course. If you do not already have a copy of Visual Basic 2005, I recommend that you obtain the Academic version of the Professional Edition. The Academic version represents a substantial discount for students and teachers.

Microsoft's website on Visual Studio 2005, <http://lab.msdn.microsoft.com/vs2005/> at the time of this writing (Microsoft does reorganize its website from time to time so this location may change), has a product matrix that lists the differences between the editions.

Installing Visual Basic 2005

Now you are ready to install Visual Basic 2005! You will find it easy.

The Visual Basic 2005 installation may consist of more than one CD, depending on the edition. It is a large program, so it takes some time to install. However, Visual Basic 2005 is not difficult to install. Installation is simply a matter of following directions and being patient. Patience is important in programming, and so it is with the installation of Visual Basic 2005.

One unusual feature is that the help for Visual Basic 2005 is not built into the program but instead is a separate program, MSDN Library. *MSDN* is an acronym for Microsoft Developer Network. This help also comes on one or more CDs, depending on the edition.

Starting Your First Visual Basic 2005 Project

Now you're going to create your first Visual Basic 2005 project. You not only will use this project for this lesson, but you also will use it as the starting point for the project in the next lesson.

NOTE *The following instructions assume you purchased Visual Studio 2005. However, the same basic information applies if you purchased Visual Basic 2005 Express Edition, though some of the screenshots may look slightly different.*

Starting the Program

Although you use Visual Basic 2005 to create programs, it itself is a program. Start Visual Basic 2005 by choosing All Programs from the Start menu, select the folder called Microsoft Visual Studio 2005, and then click on the icon of the same name that appears in the submenu.

When you first start Visual Studio 2005, a form, shown in Figure 1-1, will display, asking you to choose your default environment settings.

I chose the General Development Settings option, but you can choose the development settings for Visual Basic or one of the other programming languages. I don't consider this choice an important issue because the various settings are not that different. I chose General Development Settings because that setting is the most generic, and would work equally well if you also programmed in another language supported by Visual Studio 2005 such as Visual C#.

The Start Page next will display, as shown in Figure 1-2.

Now you are ready to start. So let's get going!

Specifying the Type of New Project

Because we want to create a new project, choose New from the File menu and then Project from the New submenu to display the New Project dialog box shown in Figure 1-3.



Figure 1-1 Choosing your default environment settings



Figure 1-2 The Start Page

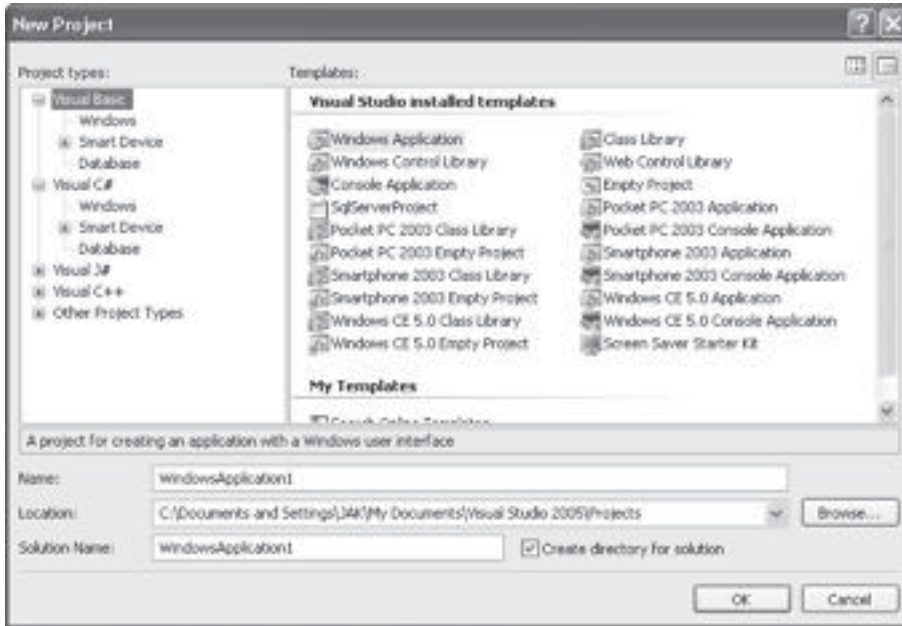


Figure 1-3 The New Project dialog box

The left pane of the New Project dialog box lists project types. There are project types for each of the languages included in Visual Studio 2005. In addition to Visual Basic, these are Visual C#, Visual C++, and Visual J#. Because this book is about Visual Basic 2005, choose Visual Basic.

The right pane of the New Project dialog box lists templates for various types of Visual Basic applications you can create. A project template helps you get started by creating the initial files, code, and other settings for the selected project.

There certainly are a lot of templates to choose from. The ones starting with Windows CE or Pocket PC can be run on handheld computers, and the ones starting with Smartphone can be run from phones. However, for most of this book, we will be creating Windows applications, so select Windows Application from the right pane. I will be discussing in Chapter 2 what a Windows application is. For now, Microsoft Word and Excel are examples of Windows applications. Each has a window or windows in which you work, with a menu, toolbar, and other visual components with which you can interact.

As shown in Figure 1-3, when you choose the Windows Application project template, the description beneath the Project Types frame becomes “A project for creating an application with a Windows user interface.”

Specifying the Name and Location of the Project

The lower part of the New Project dialog box lists the name of and location for your project. The default project name for your first project is `WindowsApplication1`. For the second, the name is `WindowsApplication2`, and so on. You should change this default name to one that will help you identify this project later. Otherwise, after you have created many projects, you may not recall what `WindowsApplication52` did as opposed to `WindowsApplication53`.

The location for your project is up to you; the default location should work fine. Whatever your decision, I recommend you have a consistent method for where you store your projects so you can easily find them later.

In Figure 1-4, I have changed the name of the project to `FirstProject` and the location of the project to another drive, `D`, on my computer.

Once you are satisfied with the name and location of the project, click **OK**. Visual Studio.NET then generates the files and folders for your first project. A folder with the same name as the project is also created in the location displayed in the Location

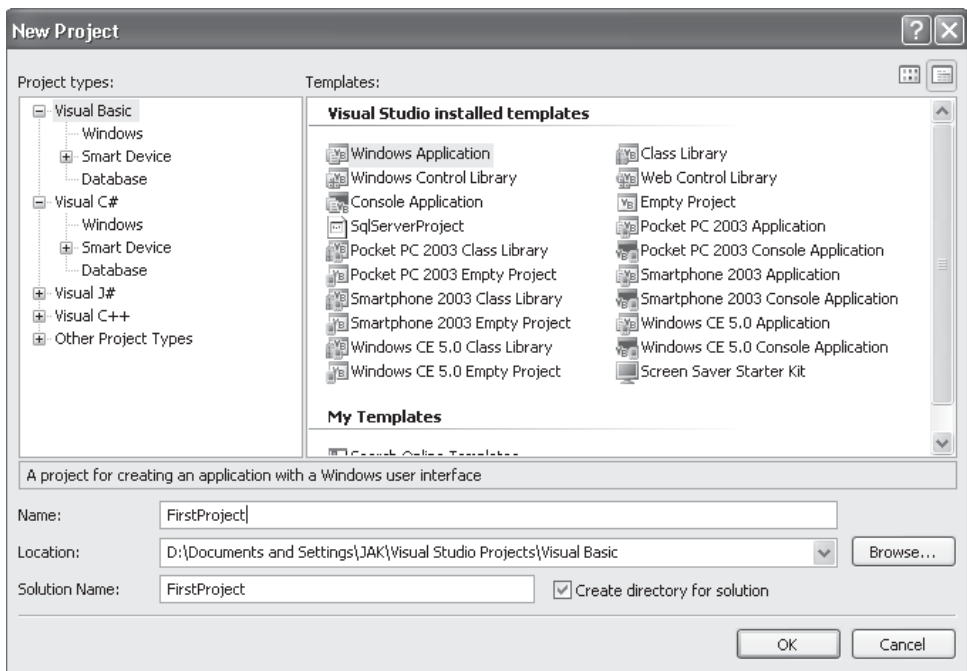


Figure 1-4 Changing the default name of and location for the project

field, which contains the parent folder where your project files will be located. Therefore, because in Figure 1-4 the project will be located in D:\Documents and Settings\JAK\Visual Studio Projects\Visual Basic, and the name of the project is FirstProject, a folder named FirstProject will be created at the specified location, and the project files will be stored at D:\Documents and Settings\JAK\Visual Studio Projects\Visual Basic\FirstProject.

Integrated Development Environment (IDE)

Figure 1-5 shows a view of the Windows application FirstProject that is created after you click the OK button in the New Project dialog box.

Figure 1-5 displays what is called an Integrated Development Environment, or IDE. The term “development environment” refers to Visual Studio.NET’s role as an application to assist you in developing applications. The term “integrated” means the tools to design your application and write, test, and run your code are all together under one (software) roof.

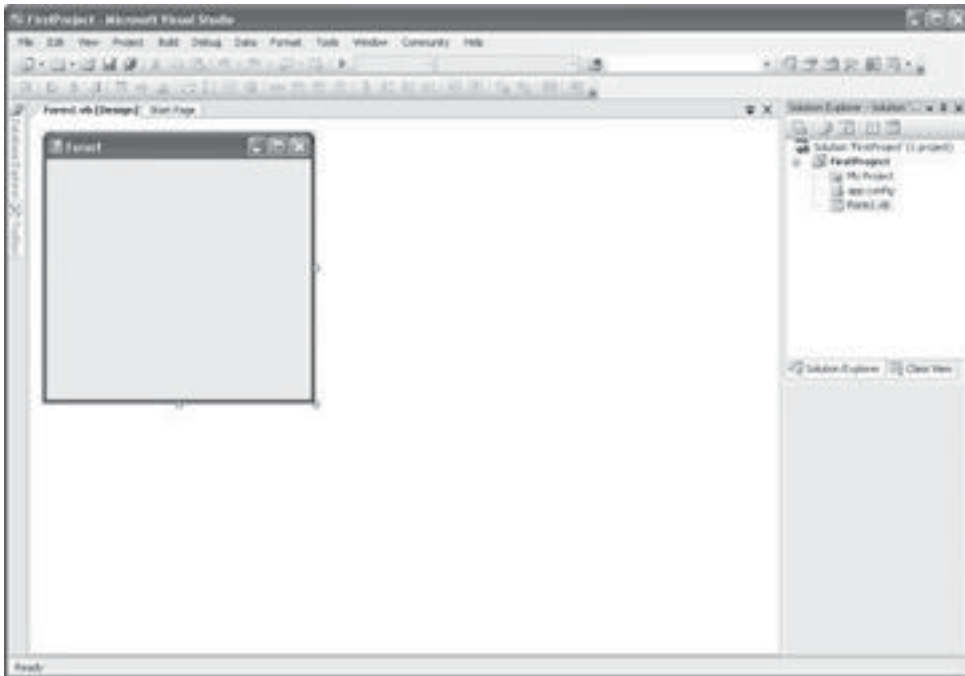


Figure 1-5 A new project

The IDE is complex, with many windows that perform many different functions. Don't worry; you don't need to know right away what they all do. Various components of the IDE will be introduced, described, and explained in this and succeeding lessons.

Run the Project!

We now will use the IDE to run the project. To run this project as an application, you must build additional files. You do so, naturally enough, from the Build menu, shown in Figure 1-6. From the Build menu, you choose one of the following four options:

- Build Solution
- Rebuild Solution
- Build FirstProject
- Rebuild FirstProject

As will be explained later in this chapter, “building” means using the compiler to translate your code into machine language the computer can understand.

NOTE *The name following Build in the third choice and Rebuild in the fourth choice is FirstProject because we changed the name of the project to FirstProject. If we had kept the default project name of WindowsApplication1, these menu items instead would be Build WindowsApplication1 and Rebuild WindowsApplication1.*

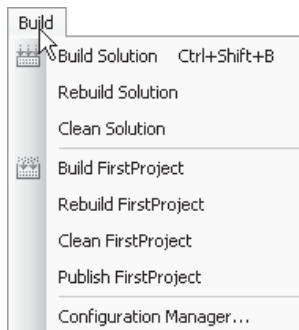


Figure 1-6 The Build menu

The difference between the Build menu items Build Solution and Build FirstProject is that the first concerns a solution and the second a project. A project contains all the files and links necessary for your application. A solution may contain multiple projects. Because the current application is simple and concerns only one project, there is no practical difference in this instance between the two menu commands.

The difference between Build and Rebuild is that if you previously have built your program, Build just builds the changes you made from the previous build, whereas Rebuild starts over and rebuilds the whole program. Rebuild consequently takes longer, so it's used when there have been extensive changes since the last build.

As a practical matter, there is little difference between the two commands. If you choose Build and the changes since the last build have been too extensive to avoid a rebuild, Visual Basic 2005 will perform a rebuild instead. The additional time a rebuild requires over a build is very minor, especially if you have a fast processor and ample RAM.

You now have a working Windows program without writing a single line of code! From the Debug menu, choose either Start or Start Without Debugging. The result is a window named Form1, shown in Figure 1-7.

The state of your project while it is running is referred to as *run time*. The state of your project before you run it, and after it stops running (such as when you click the close button of the form), is referred to as *design time*.

You now have created a working computer program. However, just what exactly is a computer program, and how does a programming language such as Visual Basic 2005 fit in? The next sections answer those questions.



Figure 1-7 A Windows application running

What Is a Computer Program?

You probably interact with computer programs many times during an average day. I certainly do. The other day, I arrived at the community college where I teach and found that my computer didn't work, so I called tech support. At the other end of the telephone line, a computer program forced me to navigate a voicemail menu maze and then tortured me while I was on perpetual hold with repeated insincere messages about how important my call was and false promises about how soon I would get through.

Finally my computer got fixed. To calm down, I decided to take a break and log on to my now-working computer to launch my favorite game program, in which community college administrators do battle with hideous alien insects from the planet Megazoid. While I was cheering on the insects, the network administrator caught me goofing off using yet another computer program that monitors employee computer usage. Fortunately, I was still employed, so an accounts payable program generated my payroll check.

On my way home I decided I needed some cash and stopped at an ATM, where a computer program confirmed (hopefully) I have enough money in my bank account and then instructed the machine to dispense the requested cash and (unfortunately) deduct that same amount from my account.

Computers are so widespread in our society because they have three advantages over us humans. First, computers can store huge amounts of information. Second, computers can recall that information quickly and accurately. Third, computers can perform calculations with lightning speed and perfect accuracy.

The advantages that computers have over us even extend to thinking sports such as chess. I used to be a professional chess player. I have not played seriously for many years, so am out of practice. However, I still was surprised that the chess program on my little Pocket PC handheld computer defeated me with ease. Even worse, the program, Pocket Fritz, taunted me in a German accent: "Dumpkopf, you have blundered again. You will now be liquidated!" My one victory was finding the mute button to silence this insolent program.

At least I have good company in defeat. In 1997, the computer Deep Blue beat the world chess champion, Garry Kasparov, in a chess match. In 2003, Kasparov was out for revenge against another computer, Deep Junior, but only drew the match. Kasparov, while perhaps the best chess player ever, is only human, and therefore no match for the computer's ability to calculate and to remember prior games.

However, we have one very significant advantage over computers. We think on our own, whereas computers don't—at least not yet, anyway. Indeed, computers fundamentally are far more brawn than brain. A computer cannot do anything

without step-by-step instructions from us telling it what to do. These instructions are called a *computer program* and of course are written by a human—namely, a computer programmer. Computer programs enable us to harness the computer’s tremendous power.

What Is a Programming Language?

When you enter a darkened room and want to see what is inside, you turn on a light switch. When you leave the room, you turn the light switch off.

The first computers were not too different from that light switch. These early computers consisted of wires and switches in which the electrical current followed a path dependent on which switches were in the on (one) or off (zero) position. Indeed, I built such a simple computer when I was a kid (which according to my kids was when dinosaurs still ruled the earth).

Each switch’s position could be expressed as a number: 1 for the on position, 0 for the off position. Thus, the instructions given to these first computers, in the form of positions on switches, essentially were a series of ones and zeroes.

Today’s computers of course are far more powerful and sophisticated than these early computers. However, the language computers understand, called *machine language*, remains the same, essentially ones and zeroes.

Although computers think in ones and zeroes, the humans who write computer programs usually don’t. Additionally, a complex program may consist of thousands or even millions of step-by-step machine language instructions, which would require an inordinately long amount of time to write. This is an important consideration because, due to competitive market forces, the amount of time within which a program has to be written is becoming increasingly less and less.

Fortunately, we do not have to write instructions to computers in machine language. Instead, we can write instructions in a “higher-level” programming language, such as Visual Basic 2005. The term “higher level” means that Visual Basic 2005 (and other languages such as C#, C++, Java, and so on) are far closer to the structure and syntax of human language than to the ones and zeroes understood by a computer. By contrast, machine language, although a programming language, is “low level” because it is far closer to the ones and zeroes understood by a computer than it is to the structure and syntax of human language. Additionally, code can be written much faster with programming languages than machine language because programming languages abstract instructions; one programming language instruction can cover many machine language instructions.

Visual Basic is but one of many programming languages. Other popular programming languages include Java, C#, and C++, and there are many more. Indeed, new languages are being created all the time. However, all programming

languages have essentially the same purpose, which is to enable a human programmer to give instructions to a computer.

There really is no one “best” programming language, but Visual Basic is an excellent choice, as for years it has been and continues to be widely used in the industry.

You may be wondering how this discussion of programming language applies since you didn’t have to write any code to achieve a working application. Although you didn’t have to write any code, that doesn’t mean code wasn’t written. Remember when you chose the project template? Visual Basic 2005 wrote code for you to create a basic Windows application.

Translating the Code for the Computer

Although you will understand the Visual Basic code you will write, the computer won’t. Computers don’t understand Visual Basic or any other programming language. They understand only machine language.

Visual Basic 2005 includes a compiler. In general, a compiler translates the code you write into corresponding machine language instructions. There are different compilers for different programming languages, but the purpose of the compiler is essentially the same, the translation of a programming language into machine language, no matter which programming language is involved.

NOTE *As discussed in more detail in Chapter 2, the compiler in Visual Basic 2005 translates the code into an intermediate language that then is translated into machine language.*

A compiler translates the code you write into corresponding machine language instructions, or into instructions that an operating system can understand and act on. However, the compiler can perform this translation only if your code is in the proper syntax for that programming language. Visual Basic 2005, like other programming languages, and indeed most human languages, has rules for the spelling of words and for the grammar of statements. If there is a syntax error, the compiler cannot translate your code into machine language instructions, and instead will call your attention to the syntax errors. Thus, in a sense, the compiler acts as a spell checker and grammar checker.

Conclusion

The way to become a good computer programmer is to write programs. To get started, you need to obtain and install Visual Basic 2005. In this chapter, you learned about the different editions of Visual Basic 2005 that are available, and how to ensure that your computer meets the hardware requirements of Visual Basic 2005. After you installed Visual Basic 2005, you learned how to use Visual Basic 2005 to create a Windows application.

This chapter then discussed what a computer program is. Computers can store huge amounts of information, recall that information quickly and accurately, and perform calculations with lightning speed and perfect accuracy. However, computers cannot think on their own; they need step-by-step instructions from us telling them what to do. These instructions are called a *computer program*, written by a human computer programmer in a programming language such as Visual Basic 2005. A compiler translates the computer program into machine language that a computer understands.

The computer program in this chapter simply displayed an empty form, or window. In the next chapter, you will examine that form further. In the process, you will learn what a Windows application is and then write your first code!

Quiz

1. What is the difference between Visual Basic 2005 and Visual Studio 2005?
2. Which operating system do you need to install and run Visual Basic 2005?
3. Which project template should you use to start creating a Windows application?
4. What is an IDE?
5. What is a computer program?
6. What is a programming language?
7. What is machine language?
8. What does “higher level” mean in the context of a programming language?
9. What does “lower level” mean in the context of a programming language?
10. What is the purpose of a compiler?

This page intentionally left blank

Writing Your First Code

When I was an elementary school student (back when dinosaurs roamed the earth, as far as my daughters are concerned), I learned through countless teacher-imposed exercises to multiply and divide several-digit numbers in my mind. Fast-forwarding more decades than I care to count, when I ask my daughters to compute the answers to less complex math homework problems, they whip out their calculators and tell me the answers—quite quickly and accurately, to be sure. When I then ask them instead to calculate the answers in their heads, they look at me as a prehistoric relic and tell me, “Aw, Dad, no one does that anymore.”

Calculators do make our lives easier. Imagine the long line at your local fast food outlet if orders had to be calculated by pencil and paper rather than with the calculators built into cash registers. In business, software programs such as Microsoft Excel enable you to perform spreadsheet calculations in minutes that might take you hours with pencil and paper.

Calculators also have a negative side effect, however. Human nature being what it is, if we don't *need* to learn something, we may decide it is not worth the time and trouble. Research suggests that the availability of calculators has contributed substantially to a decline in students' computational skills. Despite calculators, computational skills still are necessary, not just in everyday situations in which a calculator may not be available, but also as a foundation for students to develop skills in creating algorithms and analyzing problems—skills essential in, among other areas, computer programming.

Just as calculators automate computation, Visual Basic 2005 automates the creation of applications. For example, creating a Windows application strictly through code is difficult. By contrast, Chapter 1 shows that Visual Basic 2005 enables you to create a Windows application without writing a single line of code! Granted, the resulting Windows application was basic, being no more than a window with default functionality. Nevertheless, even creating such a basic Windows application strictly through code would be no small undertaking.

There is a danger of Visual Basic 2005 doing too much for beginning programmers. They may be seduced by how easy Visual Basic 2005 makes creating a Windows application. Consequently, they may just plunge in and start writing programs without really understanding the code they are writing or how the different parts of the program fit together. I have witnessed this with programming students working with prior versions of Visual Basic. They try to write more complex programs, are unable to do so because they don't understand the necessary foundation, become frustrated, and quit.

Therefore, to make a long story short (“Too late,” as my daughters would say), this chapter will explain what an event-driven Windows application is all about, including how and why the code you write executes when the user takes an action such as a mouse click. But don't worry, this chapter is not all theory. You also will put in practice what you have learned as well as write your first code!

Starting an Existing Project

Since you learn programming best by writing programs, start Visual Basic 2005. In Chapter 1 you created a new Windows application project. In this chapter, we will use that existing project instead of creating a new one. Of course, we could create a new project, but you already learned in Chapter 1 how to do that. By instead using an existing project, you will learn something new.

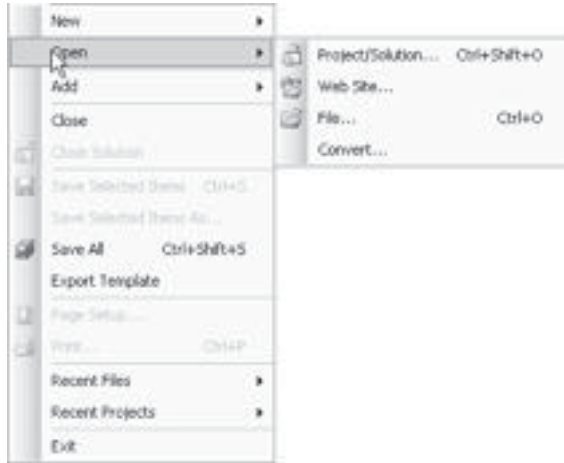


Figure 2-1 Opening an existing project

To open an existing project, choose Open from the File menu and then Project/Solution from the Open submenu, as shown in Figure 2-1. This will display the Open Project dialog box shown in Figure 2-2.

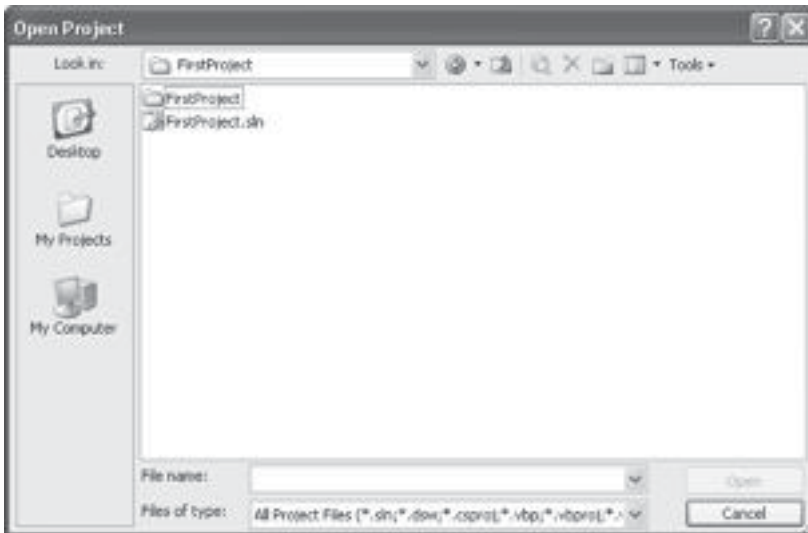


Figure 2-2 The Open Project dialog box

Using the Look In drop-down box, navigate to the folder where you saved FirstProject when you created it in Chapter 1. You then will see a file with an .sln extension, named FirstProject.sln in Figure 2-2. The .sln extension indicates a solution file. As explained in Chapter 1, a solution contains one or more projects (here, just one) used for your application.

Choose the .sln file and click the Open button in the Open Project dialog box. The Open Project dialog box will close and your FirstProject then should open, appearing as it did when you first created it in Chapter 1.

One of the windows in the project is called Solution Explorer, shown in Figure 2-3. If you don't see it, you can display it by choosing Solution Explorer from the View menu, as shown in Figure 2-4.

We will use Solution Explorer and the View menu to further examine features of this project.

Design View and Code View

You learned in Chapter 1 that the state of your program when it is running is referred to as *run time*, whereas the state of your program when it is not running is referred to as *design time*. In this section, we will be working in design time.

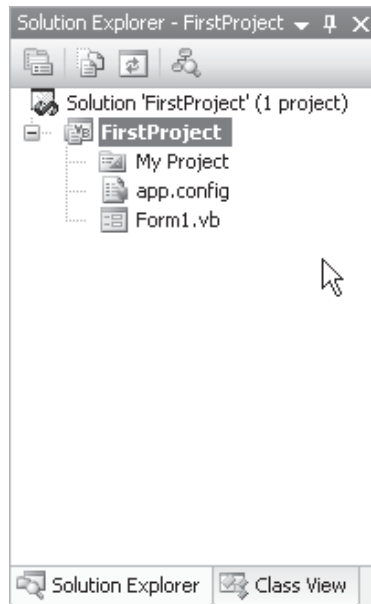


Figure 2-3 Solution Explorer

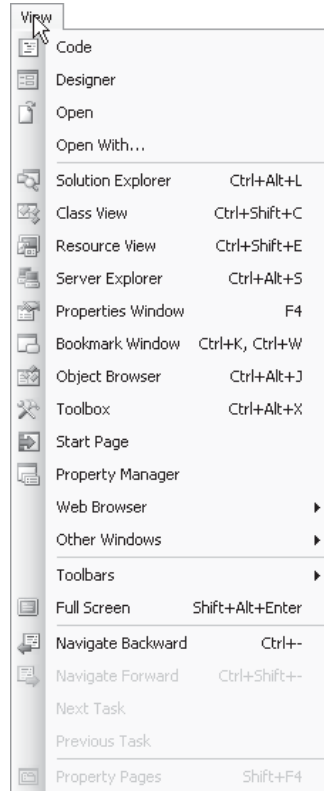


Figure 2-4 The View menu

You can view your application in two different ways during design time: designer view and code view. You choose designer view when you want to design your form, such as by resizing it or by adding to it controls such as buttons, labels and text boxes. You choose code view when you want to view or write the code of your application.

You implement designer view by first selecting `form1.vb` (the name of your form file) in Solution Explorer and then choosing Design from the View menu. An alternative is to right-click the form and choose View Designer from the shortcut menu. Either way, you will see the form, as in Figure 2-5.

You implement code view by first selecting `form1.vb` in Solution Explorer and then by choosing Code from the View menu. An alternative is to right-click the form and choose View Code from the shortcut menu. Either way, you will see code, as in Figure 2-6.

We will be working in both the designer and code views in this chapter.

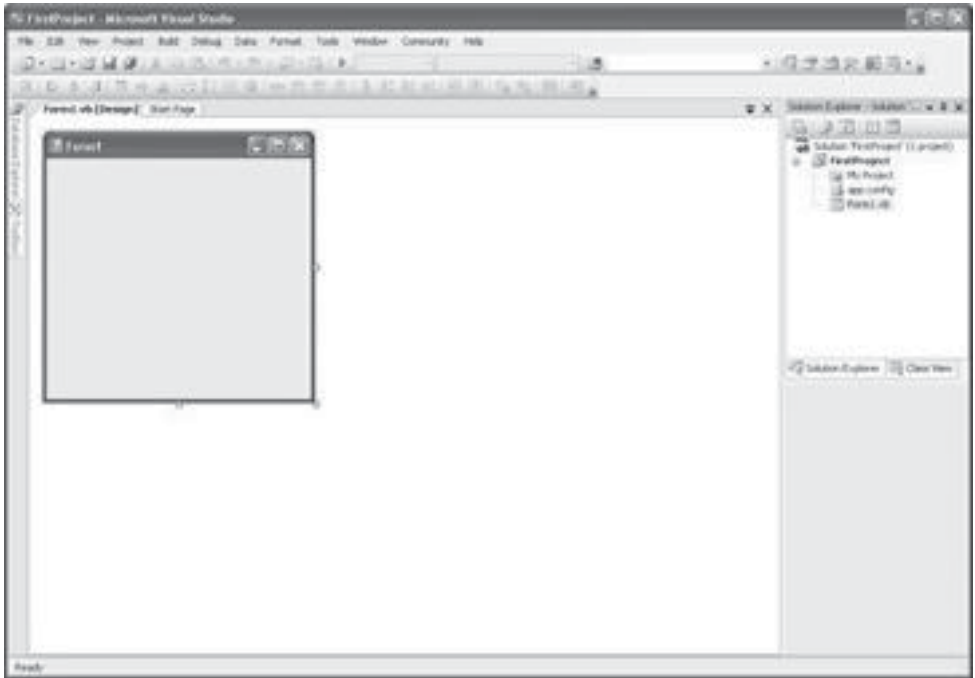


Figure 2-5 Form in designer view

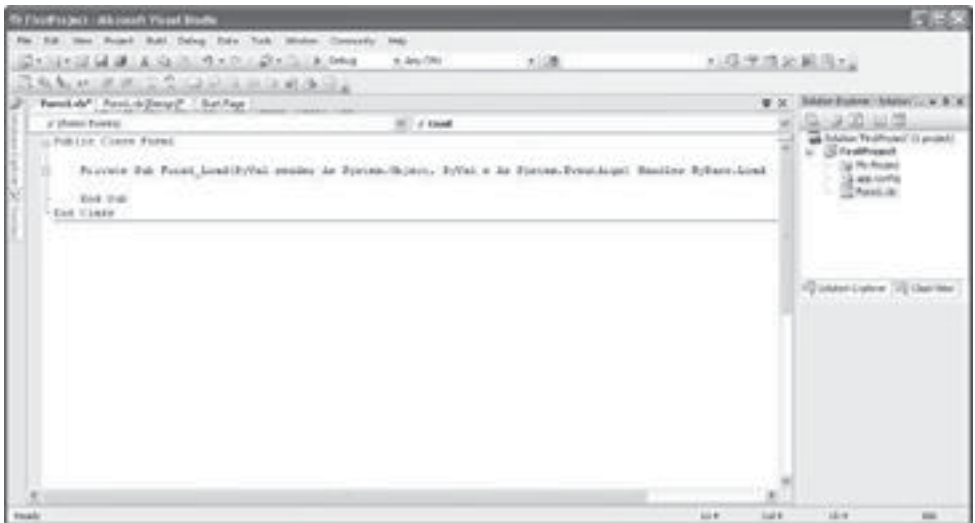


Figure 2-6 Code view

Object Browser

While in designer view, display the Object Browser, either by choosing Object Browser from the View menu or by using the shortcut key F2. The Object Browser should appear as in Figure 2-7.

Click the expander (plus sign) next to FirstProject and then highlight Form1. The Object Browser then should appear as in Figure 2-8.

The Object Browser, as its name suggests, permits you to browse, or examine, objects in your project, including the form. As Figure 2-8 shows, the lower-right pane of the Object Browser refers to “Public Class Form1.” The same reference to Public Class Form1 is on the top line of the code shown in Figure 2-6. Additionally, the lower-right pane of the Object Browser indicates that Public Class Form1 “inherits” from System.Windows.Forms.Form.

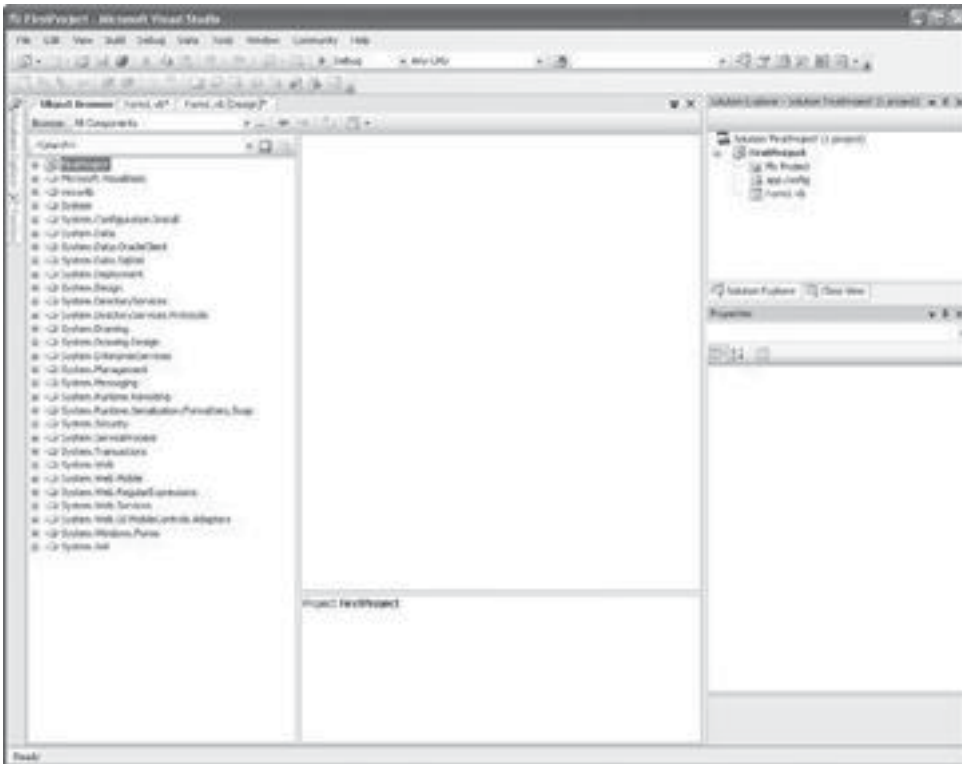


Figure 2-7 Object Browser

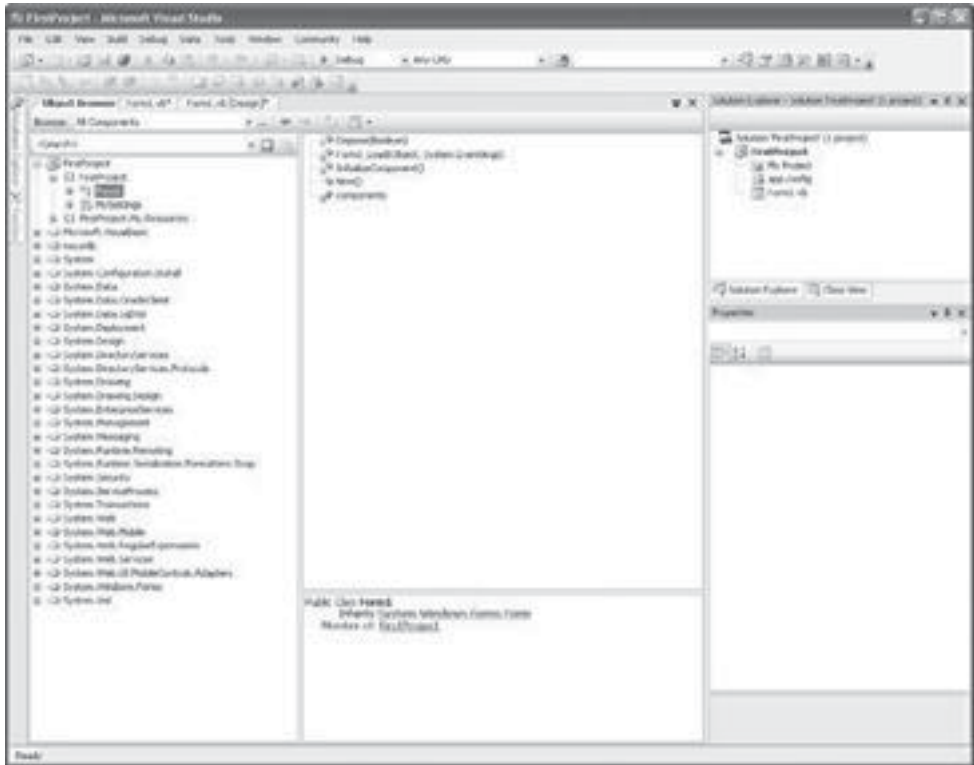


Figure 2-8 Object Browser showing information on Form1

What this terminology means is important in understanding how your first project and your future Windows application projects work. Therefore, let's now discuss this terminology.

Classes and Objects

Most programs keep track of information that relates to persons, places, or things in the real world. Such information often is complex, consisting of numerous items. For example, each of my readers is a person, and as such share certain characteristics common to all persons, such as a name, height, weight, gender, age, and so forth.

Programming languages, including Visual Basic, use classes to represent a person, place, thing, or concept. Thus, in programming parlance, each of us is an object of the Person class. A class is a pattern or template for an object, and an object is an instance of a class.

To illustrate, if my classroom contains 29 students and me as the teacher, there would be 30 objects of the Person class. Once again, each person's name, height, weight, gender, and age may differ from another's, but each of us in the room, being an object of the same class, Person, has certain common characteristics, such as a name, height, weight, gender, and age. The value of these characteristics likely will vary—two persons likely will have different names and heights, for example—but the two persons will share the characteristics themselves of having a name, a height, and so forth.

As another example, the form in our first project originated from the Form class. The Form class represents, not surprisingly, a form. A form has a number of characteristics, such as height, width, background color, text on its title bar, and so forth. Although all forms have these characteristics in common, the values of these characteristics may differ from form to form. Just as persons in a room may look different, so can forms. Some forms may be short and wide and have a blue background, and others may be tall and thin and have a yellow background. However, each of these different-looking form objects is created from the same Form class.

Inherits

The actual name of the class of the form in our application is not Form, but Form1. The Form1 class *inherits*, or starts out with, all the characteristics of the Form class. However, we can customize the Form1 class, even adding characteristics. We won't do that now, but we could.

Namespaces

As the lower-right pane of the Object Browser in Figure 2-8 indicates, the actual name of the Form class is System.Windows.Forms.Form. This means that the Form class is part of the System.Windows.Forms namespace.

To explain a namespace, let's make an analogy to the taxonomy of life you may have learned about in a biology class. All life is organized into separate kingdoms, the most commonly known being Animalia for animals and Plantae for plants. The animal kingdom is organized into several phylums, including Chordata for vertebrates. The vertebrates in the Chordata phylum are organized into several kingdoms, including Mammalia for mammals. The mammals in the Mammalia kingdom belong to different orders, including Primates for primates. Primates are subdivided into different families, including Hominidae, which in turn are subdivided into different genera, including Homo, which finally are subdivided into species, including Homo sapiens. Thus, while in biology humans generally are referred to

just by their species name, *Homo sapiens*, that species belongs to the `Animalia.Chordata.Mammalia.Hominidae.Homo` namespace.

Similarly, the `Form` class is part of the `System.Windows.Forms` namespace. The “Windows” in the namespace name stands for Windows applications.

One purpose of using namespaces is to organize code in a hierarchal manner. Another purpose is the ability to use the same class name, but in another namespace. For example, there is another `Form` class in the `System.Web.UI.MobileControls` namespace. This namespace is used for forms in web applications accessed by mobile devices, such as Pocket PCs. By contrast, the `Form` class in the `System.Windows.Forms` namespace is used for Windows applications that run from desktop or laptop computers. Both classes have the same name, `Form`, but may do so because each belongs to a different namespace.

.NET Framework

The `Form` class and the `System.Windows.Forms` namespace are defined in the .NET Framework. You will see references to the .NET Framework and .NET throughout this book, so this would be a good time to briefly explain what these terms mean.

.NET is the name for Microsoft’s strategy of software that is independent of a particular operating system or hardware. With respect to hardware, .NET projects are not limited to the traditional desktop computer. Instead, as you may recall from Chapter 1, the available templates for a Visual Basic project include ones that can be run on handheld computers or phones. Visual Studio is a tool for the development of .NET applications.

The .NET Framework consists of the Common Language Runtime (CLR) and Class Libraries. As discussed in Chapter 1, a compiler translates the code you write into machine language instructions that an operating system can understand and act on. To make a long story short, the CLR acts as a middleman between the compiler and the ultimate machine language instructions, translating intermediate language created by the compiler into the instructions. The Class Libraries include the `Form` class and the `System.Windows.Forms` namespace, as well as many other classes that we will be using in this book.

Properties

A class generally has properties. For example, the `Form` class has properties such as `Height` for its height, `BackColor` for its background color, and `Text` for text on its title bar. Therefore, objects created from the `Form` class have these properties.

Similarly, objects created by classes that inherit from the Form class, such as the Form1 class, also have these properties.

Different classes may have some properties in common. For example, the Form class has a Height property, as would a Person class. However, often one class will have a property another does not. For example, a Person class may have an EyeColor property, which the Form class does not have, whereas the Form class has a MinimizeBox property (pertaining to the minimize button at the upper right), which a Person class would not have. At least I have never seen a person with a minimize button!

Properties Window

While in designer view, choose Properties Window from the View menu. This will display the Properties window, as shown in Figure 2-9.

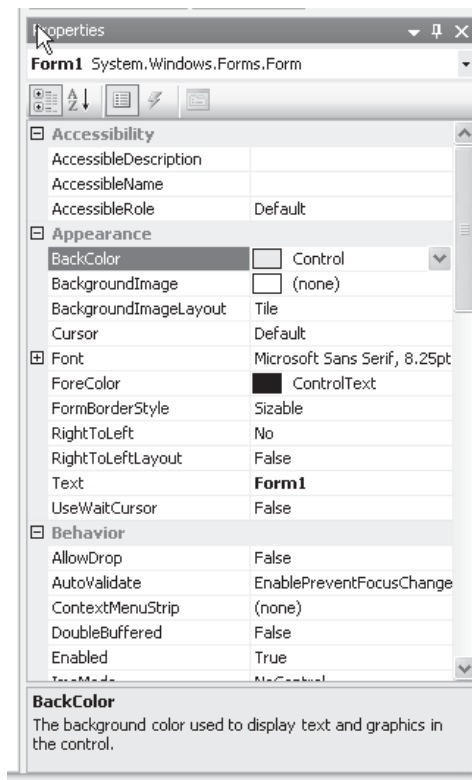


Figure 2-9 The Properties window

The Properties window lists various attributes or characteristics of the form, such as its height and width, background color, the text that appears in its title bar, and so forth. These attributes or characteristics, also referred to as *properties*, are listed in the left column. The values of these properties are listed in the right column. For example, in Figure 2-9, the value of the Text property is Form1, which is the text that appears in the title bar of the form in Figure 2-5.

The first button sorts the properties by category. This is the view in Figure 2-9. The second button sorts the properties in alphabetical order. This is the view in Figure 2-10. Don't worry about the other three buttons for now.

Many of the properties in Figures 2-9 and 2-10 have values. You did not assign those values to those properties. Rather, the IDE assigned those values because the form needs some background color, size, and so forth when you first create the application. These IDE-assigned values are referred to as *default values*. "Default" in this context refers to a property's value if you do nothing to change that value.

However, as the next section discusses, you may change default values.

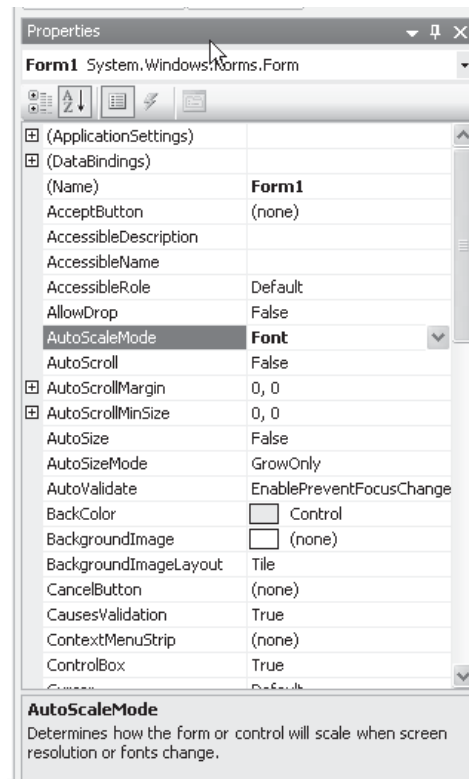


Figure 2-10 Properties listed in alphabetical order

Changing Properties at Design Time

You can use the Properties window to view the properties of the form object in your first project. You also can use the Properties window to change the value of properties of that form object at design time. For example, in the Properties window, change the value of the Text property to MyForm or some other name, and then press ENTER. The text in the form's title bar will change to MyForm or whatever other text you typed.

However, you cannot use the Properties window to change the value of properties of the form object at run time. Instead, you need to write code to change the value of properties of the form object at run time. You will learn in this chapter how to do that. However, before we get there, let's first discuss what a Windows application is, because the answer will help you understand the code you will be writing.

What Is a Windows Application?

Nowadays the majority of applications are written for at least one if not more of the Windows operating systems, which include Windows 9x, NT, 2000, XP, and 2003. Figure 2-11 shows a familiar Windows application, Notepad, which is included by default in the installation of all Windows operating systems.

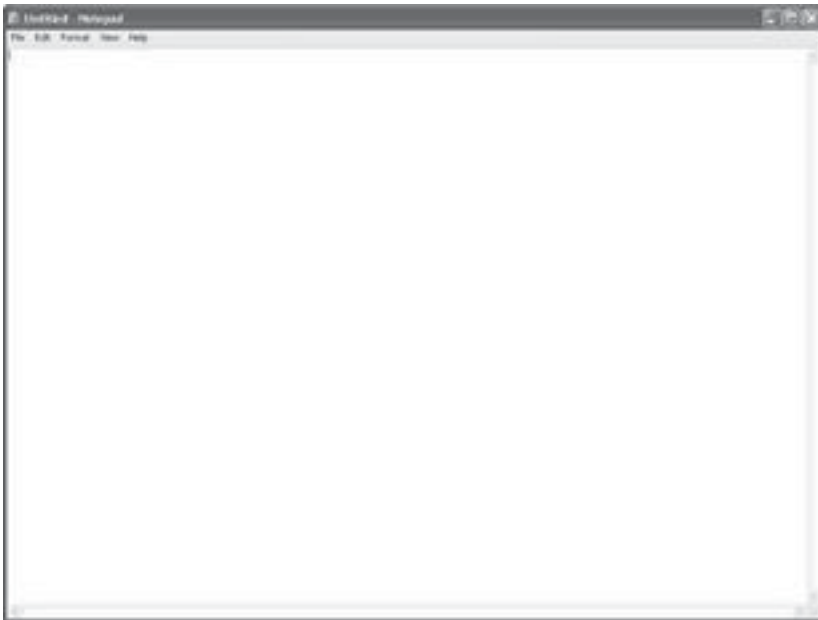


Figure 2-11 Notepad, a Windows application

Although the Windows operating system has virtually taken over the computer world, it has not been with us that long. Windows was not introduced until 1985, more than 20 years after the introduction of BASIC, the ancestor of Visual Basic, and did not catch on until the introduction of Windows 3.0 in the early 1990s. Prior to the 1990s, applications often ran in the DOS operating system. Figure 2-12 shows a DOS text editor, the DOS equivalent of Notepad in the Windows operating system. A comparison of the DOS text editor in Figure 2-12 and Notepad in Figure 2-11 shows that DOS applications have a decidedly different and less rich appearance than Windows applications.

The difference between DOS and Windows applications is more than skin deep. They also behave very differently. Let's now look at both differences.

Windows Applications Are Goopy

The hallmark of a Windows application is that the application is displayed in ... you guessed it, a window. However, there is more to a Windows application than a window. A Windows application has a graphical user interface, which is often referred to by the acronym GUI, pronounced "goopy."

A GUI usually includes a menu, such as the File, Edit, Format, View, and Help menus in Notepad (refer to Figure 2-11). The DOS text editor in Figure 2-12 also includes a menu. However, a GUI is not limited to a menu, and normally includes other visual components, such as buttons to click, edit boxes in which to type text, and so on. DOS applications have few of these other visual components.



Figure 2-12 A console window for a DOS application

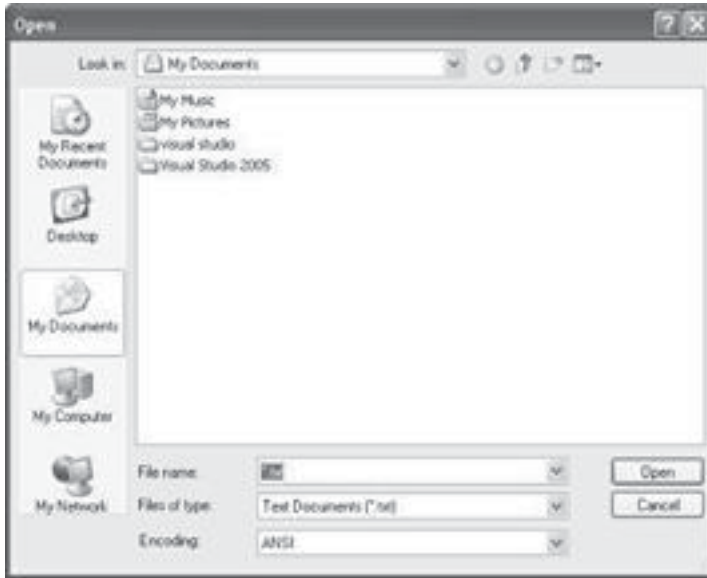


Figure 2-13 The Open dialog box in Notepad

The GUI makes Windows applications prettier than console applications, but it serves a more important purpose, which is to make Windows applications easier to use. For example, the menu in Notepad makes it easy for you to open a file. Clicking the File menu and then the Open submenu displays another visual component, the Open dialog box (shown in Figure 2-13), from which you simply pick the file you want to open.

Figure 2-14 shows the Open dialog box in the DOS text editor. This Open dialog box is far clumsier to use than the Windows counterpart in Figure 2-13.

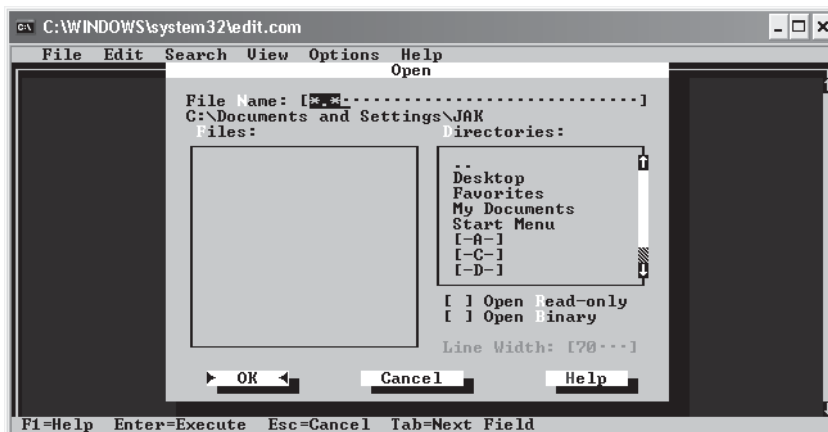


Figure 2-14 The Open dialog box in the DOS text editor

Of course, nothing is free in this world. The pretty GUI of a Windows application comes at a programming price. Code, lots of it, some of it rather complex, is required to create a window, not to mention to create the menu and other controls in the window.

This is where Visual Basic once again eases your task. You do not need to write copious, complex code to create a window. Instead, Visual Basic creates the window for you when you start a new Windows application project, and it also writes the code necessary to make that window work. This spares you substantial grunt work.

Windows Applications Are Event-Driven

Windows applications behave differently as well as look different than their predecessors.

Before Windows, applications often told the user what to do. For example, an application may tell the operating system to print to the screen the text message “Enter your name.” The user would then input their name and press the `ENTER` key. The user could not have entered their name before this point, and they had to enter data at this point or the program would not continue. The program then may tell the operating system to print to the screen “Enter your age.” The user would then input their age and press the `ENTER` key. Once again, the user could not have entered their age before this point, and had to enter data at this point or the program would not continue. Finally, the program may tell the operating system to output to the screen a sentence that includes the name and age entered, followed by whether the user is a minor, adult, or senior citizen, based on the age that was entered. The program input and output might look like this:

```
Enter your name: Jeff
Enter your age: 53
Jeff, age 53, you are an adult.
```

In this application, often called *procedural programming*, the application, not the user, determines the order in which things happen. However, Windows applications are just the opposite; the user tells the application what to do. What happens next after you open Notepad? The answer is, “It depends.” Specifically, it depends on what you, as the user, do next. If you click the `File | Open` menu item, the Open dialog box will display as shown previously in Figure 2-13. If instead you click the `Help | Help Topics` menu item, Notepad Help will display. Of course, you may decide you’re tired of Notepad and close it by using the `File | Exit` menu item or the close button. Thus, in a Windows application, the user’s actions, not the application, determine the order in which things happen.

A procedural program can be analogized to a recipe. The program follows the instructions step by step. By contrast, a Windows application can be analogized to a paramedic. The paramedic waits for a call. When a call comes, the paramedic takes the equipment warranted by the call and goes to the location. When finished, the paramedic returns to his or her station and waits for the next call, and when it comes, takes the equipment warranted by that call and goes to the next location.

In the parlance of Windows programming, the user's actions create *events* that cause the operating system to send messages to the application. For example, the user's act of clicking Notepad's File | Open menu item is an event that causes the operating system to send a message to the Notepad window that the File | Open menu command has been clicked. When Notepad receives that message, code in Notepad displays the Open dialog box. Because the events resulting from the user's actions drive the application, Windows programming often is referred to as being *event-driven*.

Classes Have Events

An event does not exist by itself. Rather, an event is something that happens to an object, usually as the result of user interaction with the object, such as its being clicked. For example, when the user clicks Notepad's File | Open menu item, the event is a click, and the object that is the subject of the event is the File | Open menu item.

The File | Open menu item is an object that is created from a class. That MenuItem class, and classes generally, have events in addition to having properties. For example, a form object has a Click event that occurs when the user clicks the mouse on the form.

As with properties, different classes may have some events in common, but usually would not share the exact same set of events.

Creating an Event Procedure

As discussed in the section "Windows Applications Are Event-Driven," you write code so the user's action in clicking the File | Open menu item in Notepad will display an Open dialog box that permits the user to choose and open a file. You want this code to execute when, and only when, your application's user clicks the File | Open menu item. You use an event procedure to solve this problem, by associating the code that displays the Open dialog box with the Click event of the File | Open menu item object. The event procedure connects the mouse click of the File | Open menu item to the code you want to run when the menu item is clicked.

When the .NET Framework that underlies Visual Basic 2005 detects an event such as a mouse click that happens to an object such as the menu item, it searches for an event procedure that matches the object and event. If the .NET Framework finds such an event procedure, it calls that event procedure, and the code inside the event procedure executes.

In this section, we will write code that will change the text displayed in the form's title bar when you click the form. To accomplish this, we need to write code for the Click event procedure of the form.

Writing code for an event procedure involves two steps. The first step is to create the event procedure stub. As will be illustrated in the next section, an event procedure stub is how the event procedure appears before you write any code. Your writing code inside that event procedure code is the second step.

Creating an Event Procedure Stub

To start creating an event procedure stub, go to code view as shown in Figure 2-6. Click on the left drop-down box and choose (Form1 Events), as shown in Figure 2-15.

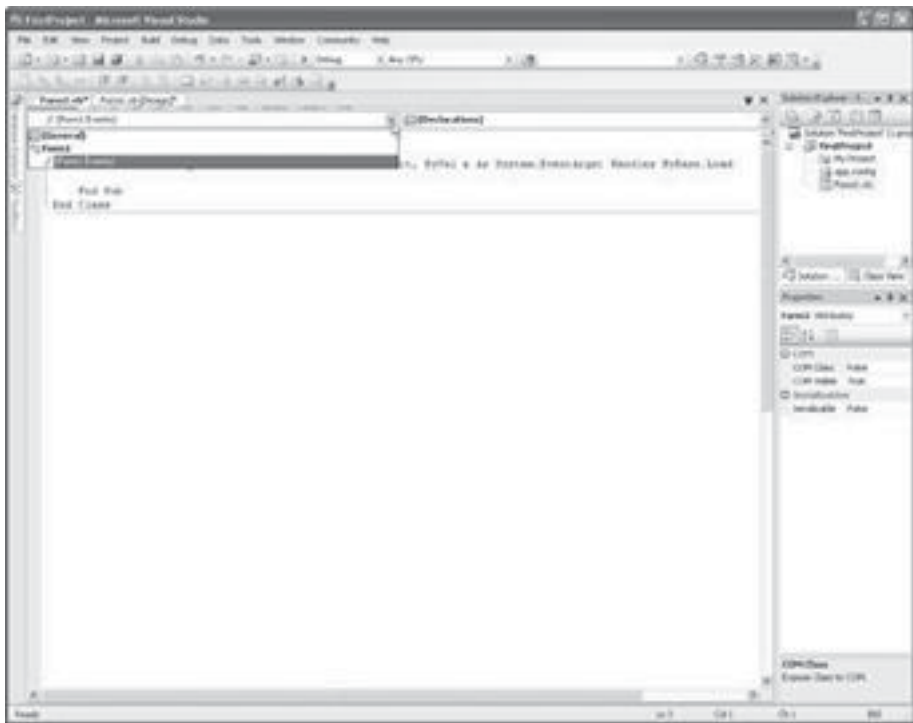


Figure 2-15 Choosing the Form1 class's events

Choosing (Form1 Events) from the left drop-down box enables you next to choose an event of Form1 from the right drop-down box. To choose a Form1 class event, click on the right drop-down box. This displays all the events of the Form1 class, as shown in Figure 2-16.

Choose the Click event from the right drop-down box. As shown in Figure 2-17, this creates an event procedure stub for the Click event of the Form1 class.

The event procedure stub is shown here:

```
Private Sub Form1_Click(ByVal sender As Object,  
    ByVal e As System.EventArgs) Handles Me.Click  
  
End Sub
```

NOTE *The first two lines actually should be one line in the code window; otherwise you will receive an error when you attempt to compile the code. That line is split into two lines in this text because of the limitation of how many characters may appear in a single line of text on the printed page.*

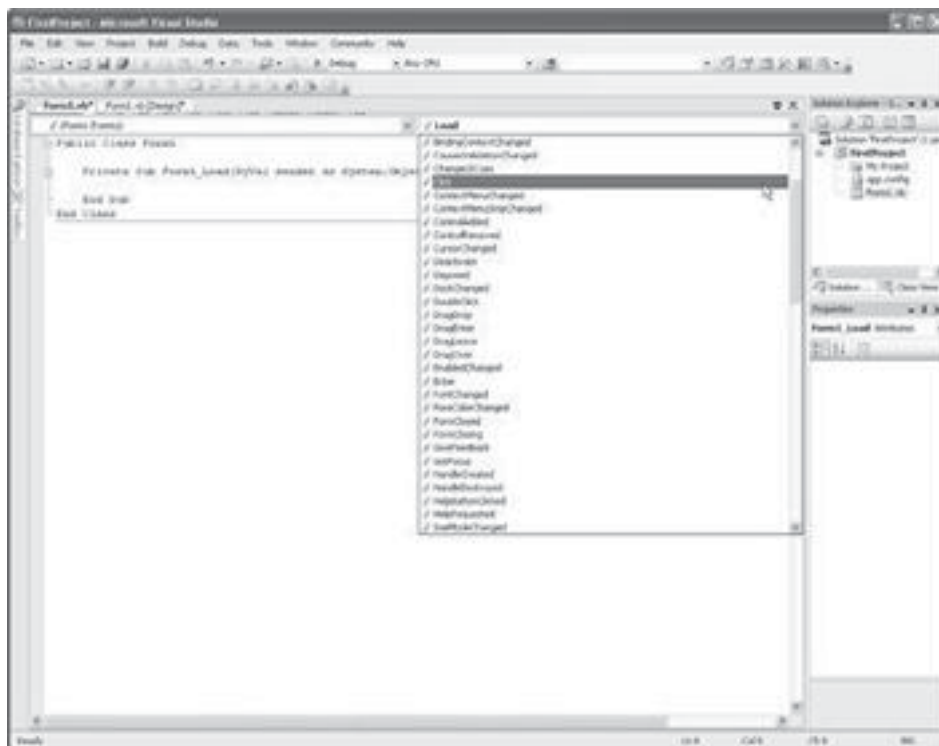


Figure 2-16 Listing of the Form1 class's events

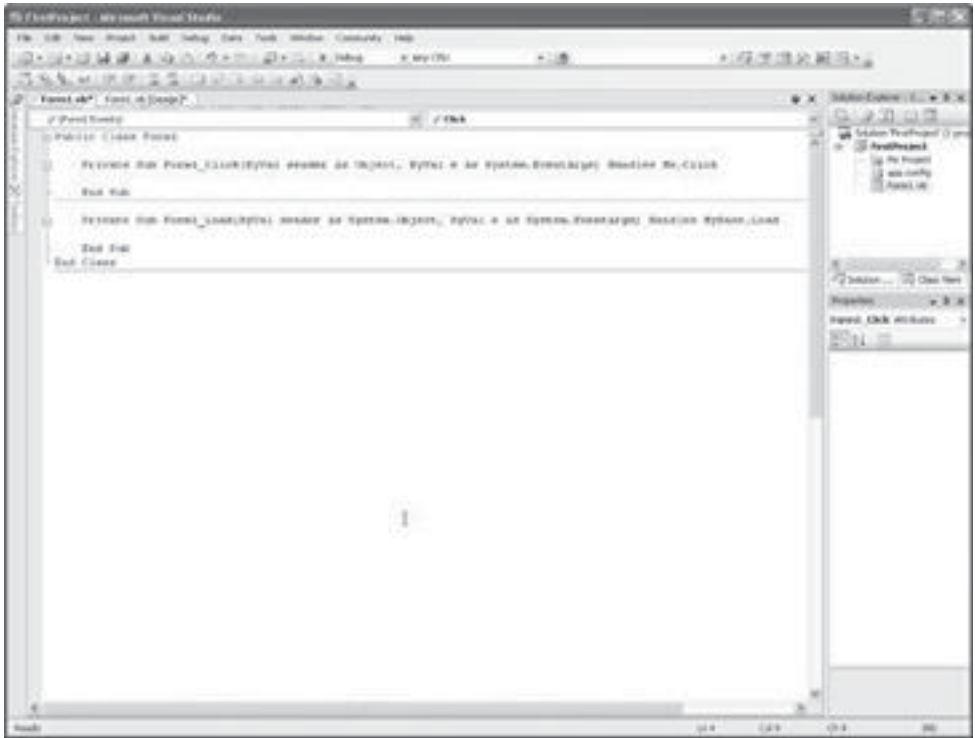


Figure 2-17 Event procedure stub

The first line of code (two lines in the text) begins the event procedure and is the title of the event procedure. It includes the name of the class object (Form1) and the name of the event (Click), separated by an underscore (Form1_Click). Don't worry about the rest of the first line of code for now; we'll cover this more later in the book.

The last line of code, End Sub, marks the end of the event procedure.

The code you will write goes between, naturally enough, the two lines. The next section discusses writing that code.

Writing Code Inside the Event Procedure

The second step is to write code inside the event procedure that will change the text displayed in the form's title bar when you click the form. Type the following code inside the event procedure:

```
Me.Text = "Eat at Joe's"
```

This code will be explained in the next section on the assignment operator.

Now your event procedure should read as follows:

```
Private Sub Form1_Click(ByVal sender As Object,  
    ByVal e As System.EventArgs) Handles Me.Click  
    Me.Text = "Eat at Joe's"  
End Sub
```

NOTE *I indented the code. This is not necessary, but it's a good habit, for reasons that will become more apparent as your code becomes more complex. Often the IDE will indent the code for you.*

Run the project by choosing Start Without Debugging from the Debug menu, as shown in Figure 2-18.

When the form first appears, the text in its title bar is the same as the value of the Text property shown in its Properties window. Now click on the form. The text in the form's title bar now should change to "Eat at Joe's."

Assignment Operator

Now that you have confirmed that the code does what it is supposed to do, I will now explain the code, which again is

```
Me.Text = "Eat at Joe's"
```

What looks like an equals sign (=) in the middle of the code is not an equals sign at all. Instead, it is called an assignment operator.

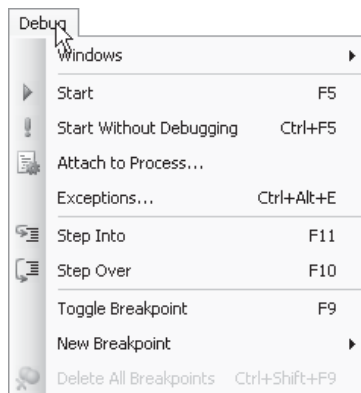


Figure 2-18 Running the project from the Debug menu

To the right of the assignment operator are words inside double quotation marks. This is called a *string*. A string usually consists of two or more characters. Characters may include a letter, a digit, a punctuation mark, or a space. The double quotation marks indicate a string; numeric values are not placed inside double quotation marks.

To the left of the assignment operator is the `Me` keyword, which is a reference to the current `Form1` object, and `Text`, a property of that object, separated by a dot or period. The code `Me.Text` thus refers to the `Text` property of the current `Form1` object.

The purpose of the assignment operator is to assign the value on its right to the property on its left. Thus, the string “Eat at Joe’s” is assigned to the `Text` property of the current `Form1` object.

This code, being inside the `Click` event procedure of the form object, executes (or runs) when, and only when, the form is clicked. When the form is clicked, the string “Eat at Joe’s” is assigned to the `Text` property of the current `Form1` object, and therefore appears in the title bar of the form.

Comments

Change the line of code

```
Me.Text = "Eat at Joe's"
```

to instead read as follows:

```
Me.Text = "Eat at Joe's"    'Changes text in title bar
```

The program will run exactly the same. In fact, the code has not changed at all. The portion of the line beginning with an apostrophe (‘) followed by “Changes text in title bar” is a comment. The apostrophe indicates that it and what follows it on the line are not part of the code, but rather a comment.

A comment is for the benefit of a programmer reading the code, the purpose usually being an explanation of the code. An explanation may not be necessary for a line of code changing the value of the text shown in a form’s title bar. However, as your applications become more complex, explanations may be helpful to fellow programmers who need to review your code. Indeed, you may find your own explanation of your own code helpful to refresh your memory if you have to return to your code months after you wrote it, either to enhance the code or to fix a problem.

Conclusion

Visual Basic, like other programming languages, represents each person, thing, or concept that is the subject of an application as a class. Objects are created, or *instantiated*, from classes.

A class, and therefore the objects created from the class, usually have properties and events. A property is an attribute of an object, such as its height. An event is something that happens to an object, such as being clicked.

A Windows application is displayed in a window that has a graphical user interface (GUI). Additionally, Windows applications are event-driven in that the user's actions, such as clicking a mouse, create events that cause the operating system to send messages to the application. You can write code that will run when those messages are received. That code is written inside an event procedure, which executes, or runs, when a specified event happens to an object.

So far the GUI of our Windows application consists only of the form itself. We will add to that GUI in the next chapter.

Quiz

1. What is designer view?
2. What is code view?
3. What is a class in a programming language?
4. What is an object of a class?
5. What are namespaces used for?
6. What is a property of a class?
7. What are characteristics of a Windows application?
8. What is an event of a class?
9. What is an event procedure?
10. What is the purpose of the assignment operator?

This page intentionally left blank



CHAPTER



3

Controls

Thus far we have focused on the Form class. The form is an important part of your application's GUI, perhaps the most important one. However, a form cannot possibly meet all the requirements of a Windows application. For example, the form does not have the functionality to permit typing of text, listing of data, selection of choices, and so forth. You need other, specialized controls for that additional functionality. Indeed, the form's primary role is to serve as a host, or container, for other controls that enrich the GUI of Windows applications, such as menus, toolbars, buttons, text boxes, and list boxes.

You will learn in this chapter how to add controls to your form using the Toolbox. You then will learn how to change the size or location of the controls on the form.

These controls, like the form itself, have their own properties, which can be changed both at design time and at run time. This chapter will provide you with guidelines on whether to assign values at design or run time in a given situation.

This chapter culminates with a project that uses a particular control, the Label control, for two purposes: first, to display data that does not change during the running of the application and, second, using event procedures, to display data that does change during the running of the application. This project also shows you how to use information, called *parameters*, available to an event procedure.

Adding Controls to the Form

I, and perhaps you too, have been requested, when first visiting a website, to fill out a registration form. Such forms may use many specialized controls. I may type my name in a `TextBox` control. I also may choose my state or country from a list supplied by a `ListBox` control. The purposes of the `TextBox` and `ListBox` controls are identified by `Label` controls displaying “Name” and “State” or “Country.” When I am finished filling in the required information, I click a `Button` control often labeled “Submit.”

Visual Basic 2005 supports many specialized controls. However, the `TextBox`, `Label`, `ListBox`, and `Button` controls are perhaps the most commonly used.

The `TextBox`, `Label`, `ListBox`, `Button`, and other specialized controls cannot exist on their own. They must be contained, or hosted, in another specialized type of control, a container control. The form is the usual choice for a container control. Indeed, the form’s primary purpose is to serve as a container or host for other controls.

Adding controls to a form through code is no easy task. Fortunately, Visual Basic 2005 enables you to easily add available controls to a form through the `Toolbox`.

Toolbox

Visual Basic uses a `Toolbox` to display controls that you can add to your form. Figure 3-1 shows the `Toolbox`, which you can display by choosing `Toolbox` from the `View` menu.

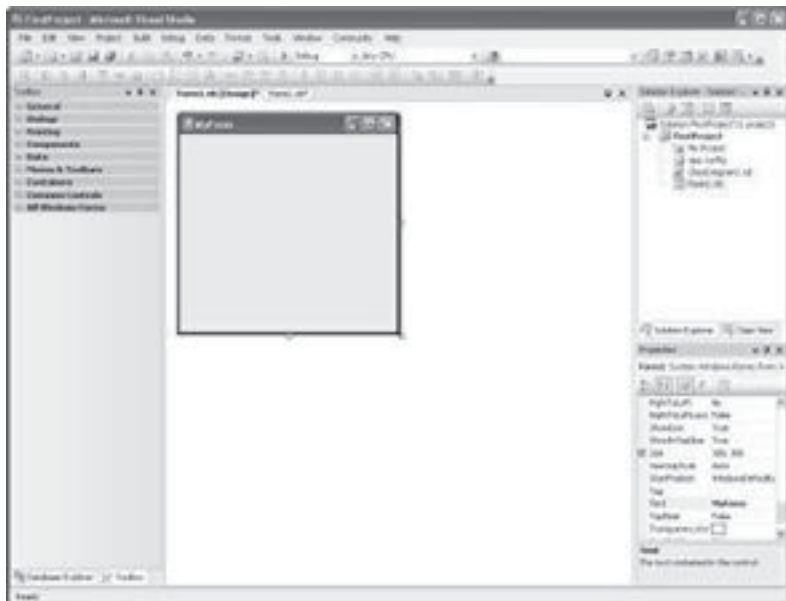


Figure 3-1 Toolbox

NOTE In following along, you can either start a new project as you did in Chapter 1 or open an existing project as you did in Chapter 2.

As Figure 3-1 shows, the Toolbox has a number of categories, each preceded by an expander (the + sign), to organize related items. If you see only the General category, the reason probably is that you are in code view rather than designer view. If so, simply switch to designer view.

The All Windows Forms category includes the controls used, naturally enough, in Windows Forms. The Common Controls category includes, as its name suggests, commonly used controls. Figure 3-2 shows the Toolbox with both categories expanded. The Label control, which we will use in the next section, appears in both categories.

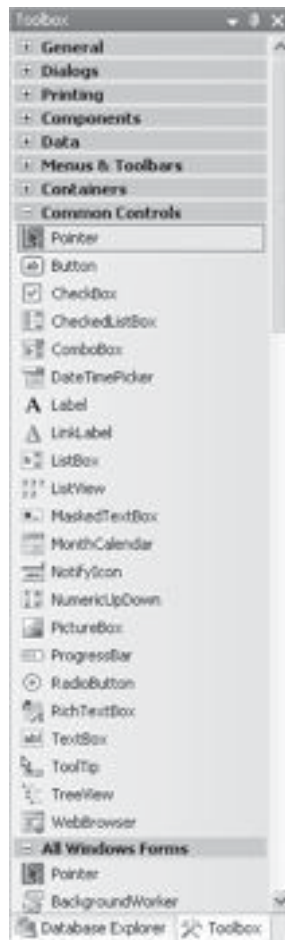


Figure 3-2 Expanding of Toolbox categories

NOTE *The Toolbox may seem to disappear if you shift focus to Solution Explorer or another part of the IDE. This is a behavior known as “auto hide.” To make the Toolbox reappear, click on the Toolbox icon on the left border of the IDE. The idea of auto hide behavior is to maximize screen space by hiding visual elements not currently in use. If you don’t want the auto hide behavior, click the pushpin button on the top of the Toolbox. Clicking the pushpin button toggles between auto hide and no auto hide.*

Copying a Control from the Toolbox to the Form

You have several methods of adding a control from the Toolbox to your form. One way is to double-click the control in the Toolbox. The control will appear somewhere in the form, such as the top-left corner. Another alternative is to click on the control in the Toolbox, drag the control over the form, and then drop the control on to the form, where the control will appear where you dropped it. Thus, with the double-click method, the IDE positions the control, whereas with the drag-and-drop method, you position the control.

Expand either the All Windows Forms or the Common Controls category to show the Label control, and then use either the double-click or drag-and-drop method to add the Label control to the form. Figure 3-3 shows the Label control after it is added to the form.

Changing the Control’s Location

As mentioned previously, the double-click method situates the Label control somewhere in the form, whereas the drag-and-drop method situates the Label control wherever you dragged and dropped it on to the form. Either way, you can reposition the Label control.

Put your mouse over the Label control. The mouse pointer should change to four arrows, as shown in Figure 3-4.

Next, click down on the left mouse button (but don’t release it) and drag the Label control to another location. Release the mouse button when the control is at the desired location.

You can also change the position of the Label control relative to the form by selecting it and then choosing either the Format | Center in Form | Horizontally menu command or the Format | Center in Form | Vertically menu command, depending on whether you want to center the control on the form horizontally or vertically (or both).

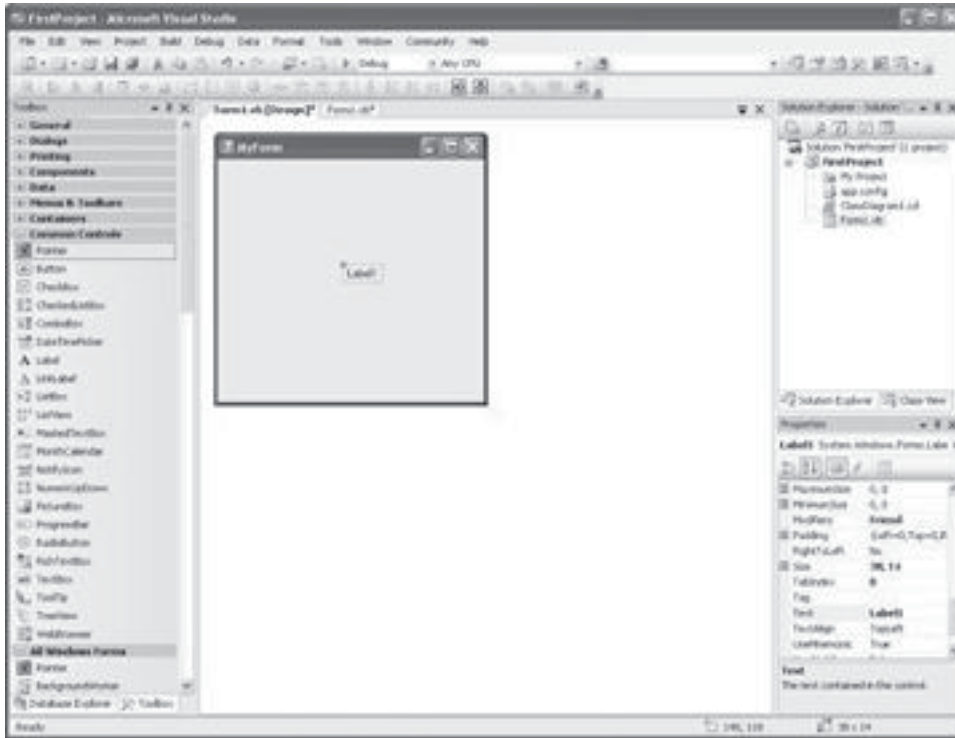


Figure 3-3 Label control inserted on the form



Figure 3-4 Mouse pointer before relocating control

If you have multiple labels, you can align the top, bottom, or sides of the controls by selecting all labels involved (click each label while holding down the **SHIFT** or **CTRL** key) and then choosing the **Format | Align |** Tops (or Middles, Bottoms, Lefts, Centers, or Rights) menu command. The label selected first (and shown with a darker highlight) will be the guide for the new alignment of all labels selected.

Changing the Control's Size

Resizing the Label control involves an extra step. The Label control has an **AutoSize** property. This property, when set to **True** (the default), automatically resizes the label so it can display its text. Figure 3-5 shows the Label control's Properties window and the **AutoSize** property.

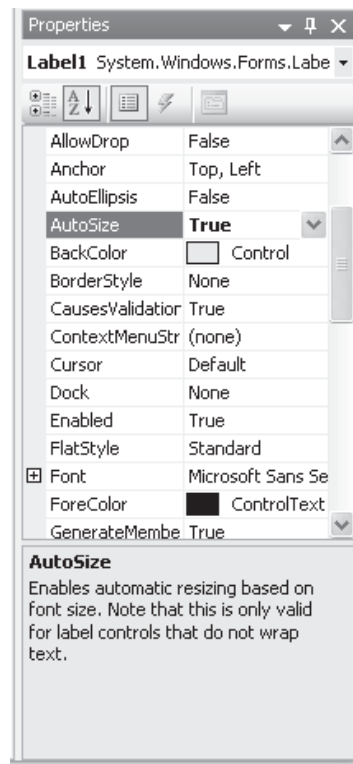


Figure 3-5 The **AutoSize** property in the Label control's Properties window

If you want to manually resize the Label control's size, you first need to set the `AutoSize` property to `False`, using the drop-down box for the value of the `AutoSize` property. Next, select the Label control you want to resize. As Figure 3-6 depicts, when you select the Label control, eight small squares appear on a box surrounding the Label control, four at the corners and four halfway between the corners.

You can resize the label by holding the mouse over one of these small boxes. The cursor should change to a two-headed arrow. Hold the mouse down and drag it to resize the label.

If you have multiple labels and their `AutoSize` properties are all set to `False`, you can make them the same width, height, or size by selecting all labels involved (click each label while holding down the `SHIFT` or `CTRL` key) and then choosing, from the `Format | Make Same Size` submenu, `Width`, `Height`, or `Both`. The size of the label selected first will become the new width, height, or size of all labels selected.

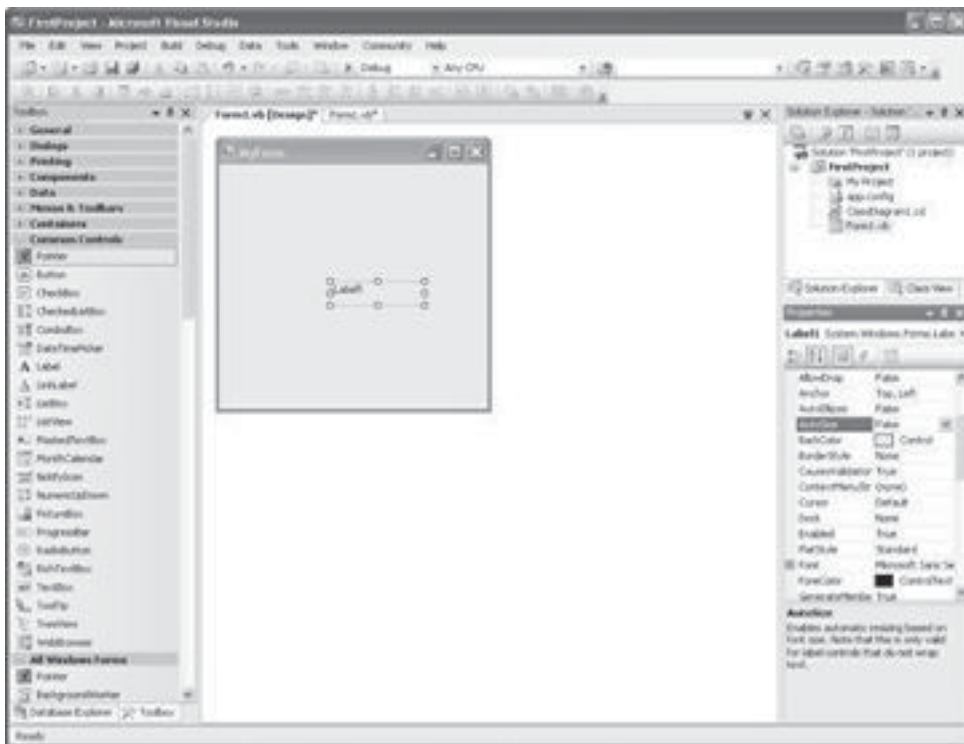


Figure 3-6 Resizing the Label control

Important Label Properties

The Label class has many properties, but the Text and the Name properties likely are the most important.

Text Property

The primary role of a label is to display text, and the value of the Text property determines the text that will be displayed.

The text is “read-only” to the application user, who cannot type on the label to change the label’s text. Other controls, in particular the TextBox control, enable the user to type on the control to change the text.

The Print dialog box shown in Figure 3-7, and displayed in most Windows applications with the File | Print menu command, illustrates two common purposes of the text in a Label control.

One common purpose of the text displayed by a label is to identify another control. In Figure 3-7, the “Number of Copies” label identifies the purpose of an adjacent control that enables you to set (with the up and down arrows) the number of copies you want to print.

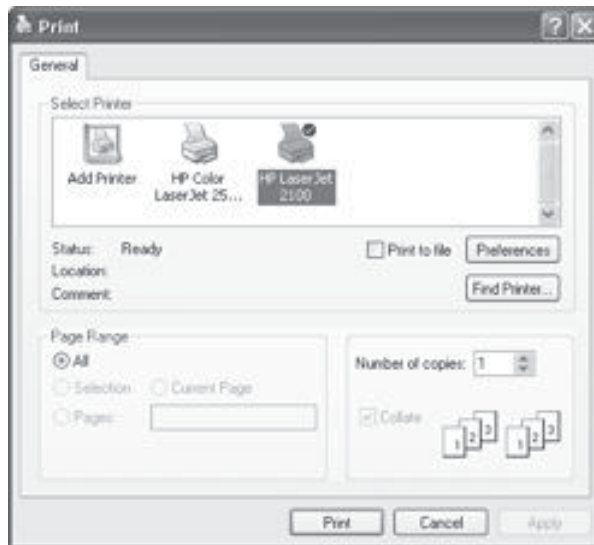


Figure 3-7 The Print dialog box

Another common purpose is to display data, such as the Label control showing “Ready” next to Status.

As with the form object, you can change the value of the Label control’s Text property either at design time or through code. You generally will use the Properties window if the purpose of the label is to identify the purpose of another control because that information usually will not change during the running of the application. The “Number of Copies” label is an example.

By contrast, you generally will use code if the purpose of the label is to display data that may change during the running of the application. For example, the Text property of the label next to Status should be set through code because, during the running of the application, the printer’s status may change between being ready and going offline.

Name Property

The Name property is important because its value is how the label is referred to in code.

By default, the first label you add to your form is named Label1, the second Label2, the third Label3, and so forth. The default name is fine if you will not be referring to the label in your code. This would be the case if the purpose of the label simply is to identify the purpose of another control.

However, using a default name can cause you difficulty if you are referring to the label in code, such as if the purpose of the label is to display information that may change when the application is running. The difficulties you may encounter increase as the number of the labels in your application increase. For example, you may have difficulty remembering if Label53 is the one that displays weather information or the one that displays your bank account balance.

I recommend you use a naming convention when naming your controls. A naming convention simply is a consistent method of naming controls. There are a number of naming conventions. It is not particularly important which naming convention you use. What is important is that you use one and stick to it.

One often-used naming convention is to name a control with a prefix, usually all lowercase and consisting of three letters, that indicates the type of control it is, followed by a word, first letter capitalized, that suggests its purpose. For example, lblWeather would indicate a label that displays weather information. If you need more than one word to describe the control’s purpose, you should combine the words into one (because a name cannot have embedded spaces) but capitalize the first letter of each word. For example, lblBankAccountBalance would indicate a label that displays your bank account balance.

Tip Be careful when you use prefixes such as *lbl* that you use a lowercase *L* and not the number *1*. Interchanging the two can cause typos that are hard for you to see, and also will result in a compiler error because control names cannot start with a number.

The Label Control in Action

In this section, you will create a project (or reuse an existing project) to display the X and Y coordinates of the mouse pointer while the mouse is moving over the form. Figure 3-8 shows what the application looks like when it is running. Of course, the X and Y coordinates displayed will vary depending on where the mouse is located over the form.

Mouse Coordinates

A brief explanation of how mouse coordinates work may be helpful before explaining how the code works. Similar to the concept of coordinates in graphing, mouse coordinates are expressed in two numbers. The first is usually referred to as X and measures a horizontal distance from a reference point. The second is usually referred to as Y and measures a vertical distance from a reference point. In the context of a mouse moving over a form, the reference point is the top-left corner of the form. Thus, the X coordinate measures the horizontal distance from the left side of the form, and the Y coordinate measures the vertical distance from the top of the form.

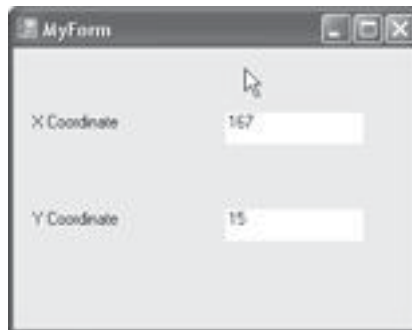


Figure 3-8 Application displaying mouse coordinates

Coordinates by convention are expressed with the syntax X,Y. Therefore, the top-left corner of the form would be the coordinate 0,0. If a coordinate is 60,77, the mouse is 60 units to the right of the left edge of the form, and 77 units below the top edge of the form.

The unit of measure is a pixel, a shortened term for “picture element,” which is a dot representing the smallest graphic unit of measurement on a screen. Screen resolutions such as 1024×768 are expressed in pixels.

Creating the Application

Implement the following steps to create the application:

1. Either open an existing project or create a new one.
2. Using the Toolbox, add four labels to the form, one label at a time.
3. Using the Properties window, change the AutoSize property of all four labels from the default, True, to False. This step will make easier the customization of the labels in the following steps.
4. Size and align the four labels as shown in Figure 3-8. The preceding sections on “Changing the Control’s Location” and “Changing the Control’s Size” explain how to align or size multiple labels.
5. Using the Properties window, change the Text properties of the two labels on the left to X Coordinate and Y Coordinate, respectively, because the purpose of these labels is to identify the two labels on the right. You are changing the value of the Text property of these labels at design time because the text on these labels will not change while the project is running.
6. Using the Properties window, change the Name properties of the two labels on the right to lblX and lblY, respectively. As discussed in the preceding section on the Name property, the prefix lbl (lowercase L, not the number 1) identifies these controls as labels to programmers reading the code, and the suffixes X and Y note the purpose of the controls—to display the X and Y coordinates, respectively. It is not so important to rename the two labels on the left because it is unlikely you will need to refer to them in code.
7. Again using the Properties window, change the BackColor property of lblX and lblY to White (so they will be more visible after we delete their text in the next step). When you click the value of the BackColor property, a tabbed dialog box appears. Choose the Custom tab and then click on a box that is white.

8. Also using the Properties window, delete any value in the Text properties of lblX and lblY so both are blank. We don't want these labels' names to display as the labels' text when the project first starts up.
9. Create an event procedure stub for the MouseMove event of the form. The process is similar to the one in Chapter 2, when you created a Click event procedure for the form. In code view, choose (Form1 Events) from the left drop-down box and then MouseMove from the right drop-down box. The event procedure stub is shown in Figure 3-9.

NOTE *The first line is too long to be displayed entirely in the window without horizontal scrolling. Later in this chapter, you will learn how to use the line-continuation character so the one long line can be divided into several shorter and more readable lines.*

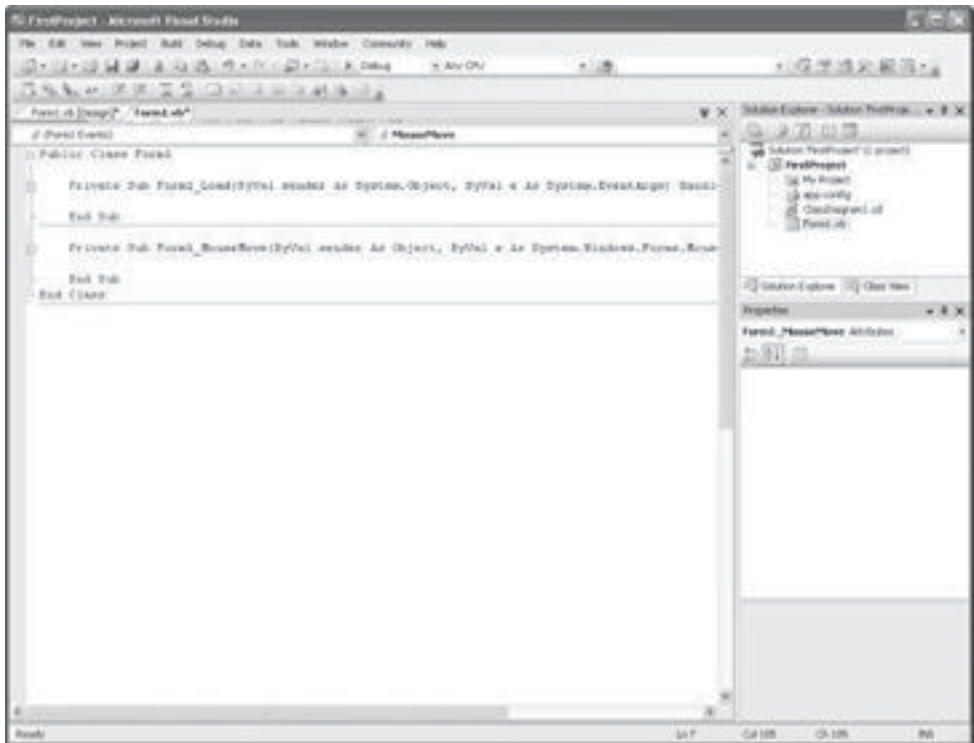


Figure 3-9 Event procedure stub for the MouseMove event of the form

10. Write the following code inside the event procedure stub:

```
lblX.Text = e.X  
lblY.Text = e.Y
```

The completed event procedure now is shown in Figure 3-10.

11. Compile the project from the Build menu and then run the project from the Debug menu. Move your mouse over the form. The two labels on the right should display numeric values, as shown in Figure 3-8, and change as you move the mouse.

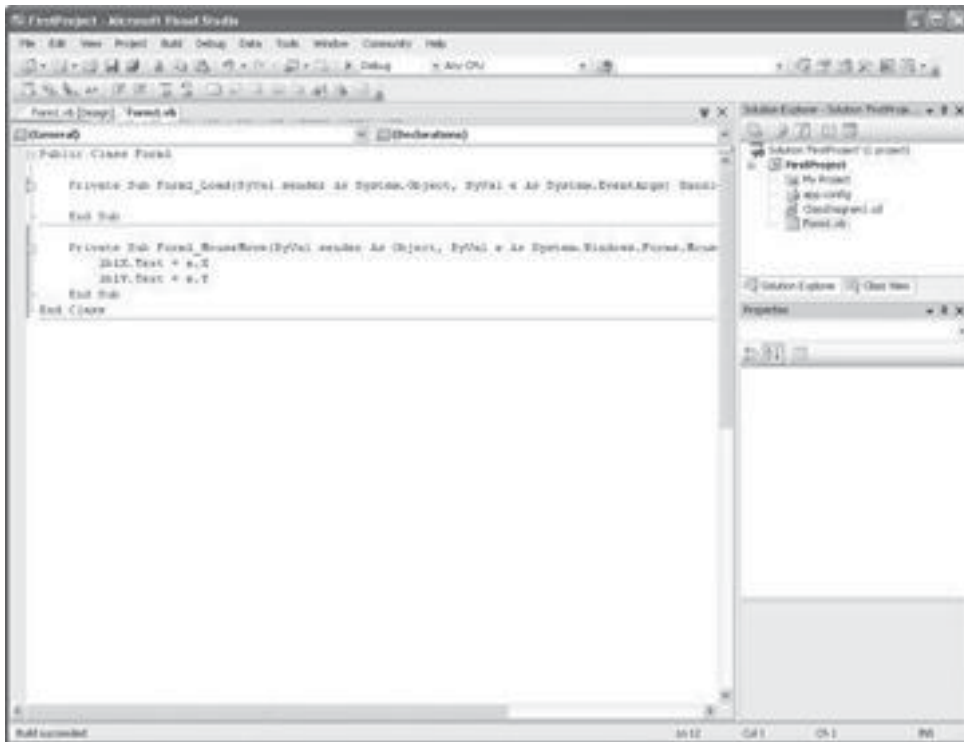


Figure 3-10 Completed MouseMove event procedure

How the Code Works

Although we know that the code works, we also need to know how the code works. However, before explaining how the code works, let's try to make that long first line of the event procedure easier to read.

Line-Continuation Character

The event procedure, after we've edited the long first line, reads as follows:

```
Private Sub Form1_MouseMove(ByVal sender As Object, _  
    ByVal e As System.Windows.Forms.MouseEventArgs) _  
    Handles Me.MouseDown  
    lblX.Text = e.X  
    lblY.Text = e.Y  
End Sub
```

The long first line now is divided into three shorter lines. This is accomplished using the line-continuation character, which is an underscore (_) preceded by a space.

Without the line-continuation character, dividing the long first line into three shorter lines would result in a compile error. The reason is Visual Basic assumes each line of code is complete. The line-continuation character tells Visual Basic that the three lines of code go together.

Using Event Procedure Parameters

The following two lines of code display the X coordinate of the mouse in the Text property of the Label control lblX and the Y coordinate of the mouse in the Text property of the Label control lblY:

```
lblX.Text = e.X  
lblY.Text = e.Y
```

The “e” on the right side of the assignment operator also appears in the parentheses of the event procedure:

```
(ByVal sender As Object, _  
    ByVal e As System.Windows.Forms.MouseEventArgs)
```

The parentheses of an event procedure contain its parameters. A parameter represents information that is available to a procedure.

An event procedure may have no parameters, one parameter, or two or more parameters. An event procedure's parameters are defined by Visual Basic and the underlying .NET Framework; you cannot change them.

When a procedure has two or more parameters, the parameters are separated by a comma. The `MouseMove` event procedure of the `Form` class has two parameters.

The second parameter, represented by `e`, is an object of the `EventArgs` class, which belongs to the `System.Windows.Forms` namespace.

The `EventArgs` class has two properties, `X` and `Y`, whose values, in the case of the `MouseMove` event, are the current `X` and `Y` coordinates of the mouse cursor. Because `e` represents the instance of the `EventArgs` class involved in the current mouse movement, `e.X` represents the `X` coordinate of the mouse when the mouse is moved, and `e.Y` represents the `Y` coordinate of the mouse when the mouse is moved. With the assignment operator, these `X` and `Y` coordinates are assigned to the `Text` properties of `lblX` and `lblY`, respectively, which then display these coordinates. Each time the mouse moves, the `MouseMove` event occurs, and therefore the code inside the event procedure executes, updating the text displayed in the two labels.

Handles Clause

The end of the first line of the event procedure is

```
Handles Me.MouseDown
```

As explained in Chapter 2, when the .NET Framework that underlies Visual Basic 2005 detects an event, such as the mouse button being held down, that happens to an object such as the menu item, it searches for an event procedure that handles that event for that object. If the .NET Framework finds such an event procedure, it calls that event procedure, and the code inside the event procedure executes.

The `Handles` keyword is used to declare that a procedure handles a specified event. That event is specified by `Me.MouseDown`. The keyword `Me` refers to the current object of the `Form1` class (that is, the form over which the mouse button is being held down). `MouseDown` is the event. Accordingly, the `Handles` clause indicates that this event procedure handles the `MouseDown` event of the form.

What If You Type the Wrong Code?

It is inevitable as you write more code that on occasion the syntax of your code will be incorrect, such as if you misspell or leave out a term. For example, instead of `Text`, you could type `Txt` so that the line of code

```
lblX.Text = e.X
```

instead is

```
lblX.Txt = e.X
```

Visual Basic 2005 tries to warn you even before you attempt to compile your code. As Figure 3-11 shows, the term `lblX.Txt` will be underlined with a squiggly line, similar to how Microsoft Word highlights misspellings.

If you hold your mouse over the highlighted code, a ToolTip shows with the following warning: “The name ‘Txt’ is not a member of ‘System.Windows.Form.Label.’” This warning means that `Txt` is not a property of the `Label` class (which is part of the `System.Windows.Form` namespace) and therefore is not recognized by the compiler. This is true because the property is spelled “Text,” not “Txt.”

Undeterred by this warning, you nevertheless attempt to build the project. As Figure 3-12 shows, an Error List should display reporting, similarly to the ToolTip, “The name ‘Txt’ is not a member of ‘System.Windows.Form.Label.’” Additionally, the line containing the error is identified.

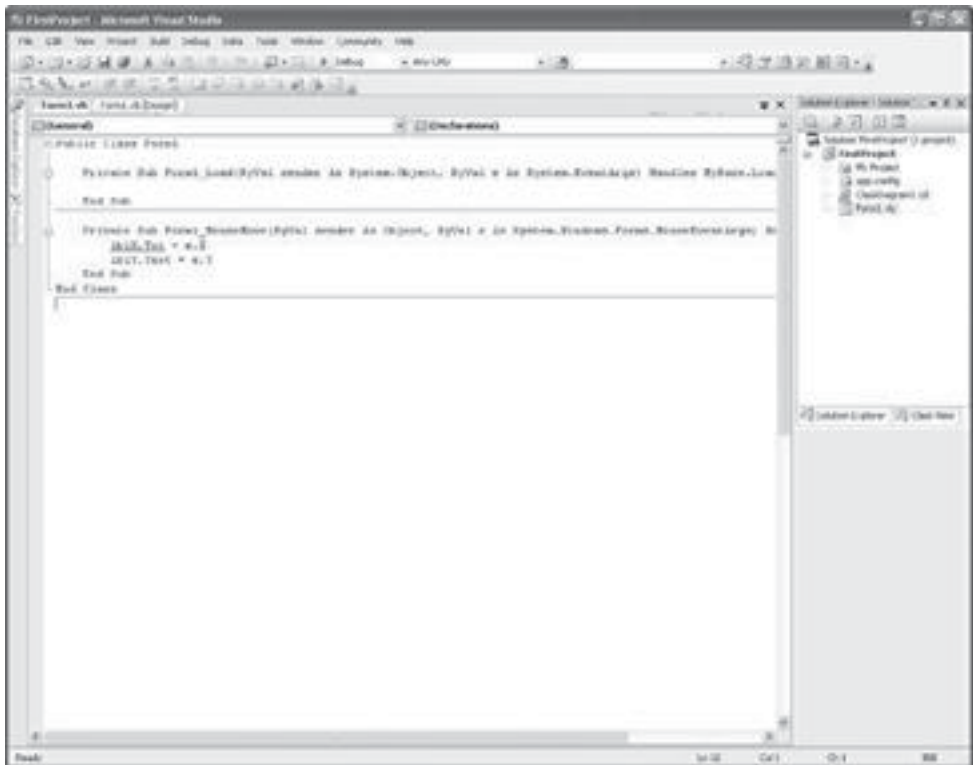


Figure 3-11 Incorrect code highlighted

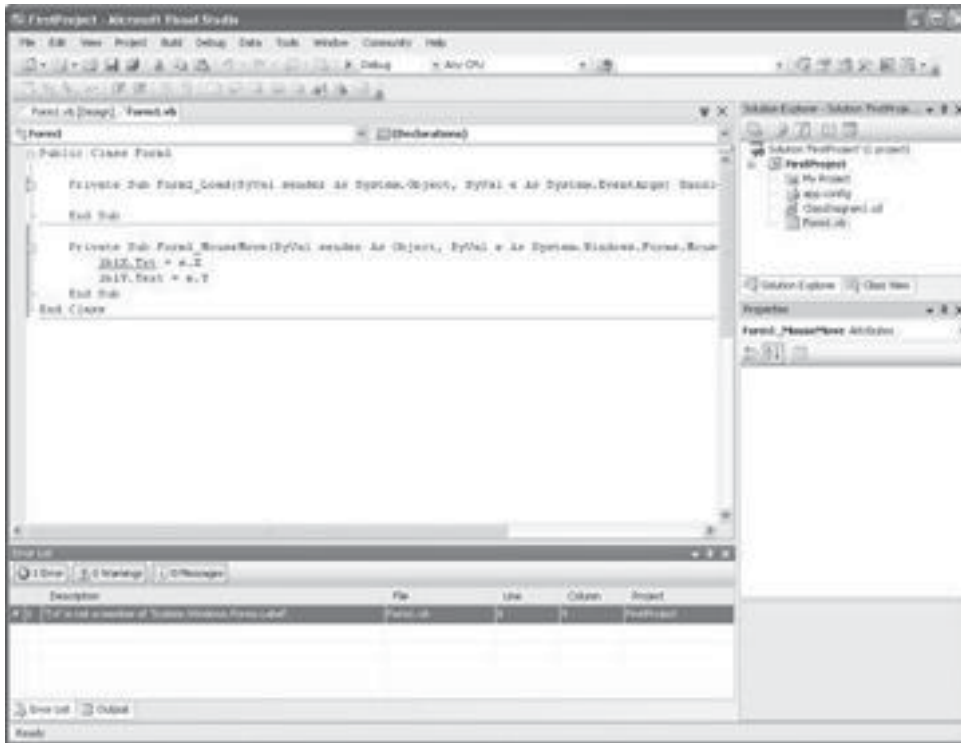


Figure 3-12 Error List reporting an error

NOTE If the Error List does not automatically display, you can display it with the menu command *View | Other Windows | Error List*.

Of course, you still need to correct the code. However, the Visual Basic 2005 IDE does advise you of the nature and location of the syntax error.

Conclusion

The form is perhaps the most important control. However, a single form without controls could only satisfy the requirements of the simplest Windows application. The form does not permit typing of text, listing data, selection of choices, and many other tasks that an application may need to perform. You need other, specialized controls for that additional functionality. Indeed, the form's primary role is to serve

as a host, or container, for controls such as menus, toolbars, and buttons, which enrich the GUI of Windows applications.

This chapter showed you how to add controls to your form using the Toolbox. You then learned how to change the size or location of the controls. The project also showed you how to control the size and location of multiple controls relative to each other.

The Label class, like the Form class, has properties. Perhaps the most important properties of the Label class are its Name and Text properties.

The Name property determines how you refer to a label in code. You should use a naming convention when naming a label that you will refer to in code. This chapter suggested a naming convention using a prefix, usually all lowercase and consisting of three letters, that indicates the type of control it is, followed by a word, first letter capitalized, that suggests its purpose.

The Text property determines the value of the text displayed by the label. Like the Text property of the Form class, you can change the value of the Label control's Text property either at design time or through code. You generally will use the Properties window if the purpose of the label is to identify the purpose of another control because that information usually will not change during the running of the application. By contrast, you generally will use code if the purpose of the label is to display data that may change during the running of the application. This code often will be located inside of an event procedure.

This chapter included a project that uses the Label control for both purposes—to display data that does not change during the running of the application and to display data that does change during the running of the application. Finally, you learned how to use information, called parameters, available to an event procedure.

Although it is impressive that you can create a working Visual Basic 2005 program that displays information using controls by writing only two lines of code, most programs need to save information, or data. The next chapter will teach you about different data types and how to create and use information storage locations called variables.

Quiz

1. What are examples of controls?
2. What is the purpose of the Toolbox?
3. How do you add a control from the Toolbox on to your form?

4. What is the purpose of the Name property of a control?
5. What is a naming convention?
6. What characteristic of the Label control does its Text property determine?
7. What are purposes of the text displayed by a Label control?
8. What is a line-continuation character?
9. What is a parameter of an event procedure?
10. What is a Handles clause?

This page intentionally left blank



PART TWO

Programming Building Blocks: Variables, Data Types, and Operators

This page intentionally left blank

Storing Information—Data Types and Variables

I often am asked for my autograph. Unfortunately, my autograph usually is requested by those who want my money, such as on credit card receipts when I purchase groceries or gas, or on checks to pay my mortgage or auto insurance.

These companies who love sending me bills could not possibly keep track of their thousands of customers using pencil and paper. Instead, they use computer programs, which harness the computer's unparalleled ability to store information and make computations using that data.

These companies are not the only ones who need to store and retrieve data. Visual Basic 2005 also needs to store and retrieve data, such as the height, width, and background color of your startup form, necessary in order for your projects to run.

Data comes in different varieties. Some data is numeric, such as the amount of my gas bill or the height of a form. Some data is text, such as my name on my gas bill or the text on the title bar of a form. Some data is Boolean (either true or false), such as whether I qualify for the senior citizen discount or whether a form is visible.

The type of information, whether numeric, text, or Boolean, is referred to as the *data type*. I will explain in this chapter the different data types and how to select the one that best fits your purpose.

You also will need to store data. Visual Basic forms and controls have many built-in properties to store data, such as the Text property of a Label or TextBox control. However, these properties are limited to storing the information they were designed for. The Height property of a form only can store a form's height, not some other information you need to store.

Visual Basic 2005 enables you to create your own information storage locations, called *variables*. I will show you in this chapter how to create and use variables.

Finally, certain values never change while a program is running. For example, if you are writing a program to calculate the cost of a transaction, the percentage of sales tax will not change while your program is running. These values that do not change while your program is running are called *constants*. I will also show you in this chapter how to create and use constants.

Data Types

Think of all the different types of information you need to keep in your mind. For example, if you as a student were driving to school for the first day of class, you would not want to be late. Therefore, you would consider the number of miles to school in deciding what time to leave. You may wonder if you will be able to get into the class and try to remember the name of the teacher you need to ask. The class will be tough, so you think about the effect the class might have on your grade point average.

Some of these items of information are numeric, such as the number of miles to school and your grade point average. However, the name of the teacher is not numeric, but text, and whether you will be able to get into the class will be either yes or no. The type of information, whether text, numeric, or yes/no, is referred to as the data type.

Numeric Data Types

Visual Basic has a number of data types—Integer being the most common—that may be used for whole numbers. A whole number may be positive (55) or negative (–55) or zero. However, the Integer data type should not be used for floating-point numbers—that is, those that have numbers to the right of the decimal point, such as –.5, 0.5, or 5.5.

The Integer keyword for this data type is an alias for the System.Int32 data type in the .NET Framework. Indeed, each of the Visual Basic data type keywords we will be discussing is an alias for a corresponding .NET Framework data type.

An Integer would be a good choice for the number of miles to school. Normally, you would think it is 8 miles to school, for example, not 8.3 miles, as there is no need to be so precise as to figure out tenths of miles.

Visual Basic has several data types, such as Double, Single, and Decimal, that may be used for floating-point numbers (for example, .5, 0.5, and 5.5). These data types would be a good choice for the grade point average, such as 3.91, because for a grade point average you want to take into account the digits to the right of the decimal point. After all, if you worked hard to earn a 3.91 grade point average, you would not want the .91 ignored, thus making your grade point average 3 instead.

NOTE *The Integer and Double data types can handle almost all numbers you may use in a program. However, there are numbers that are too large for either data type to handle, such as distances between galaxies in the universe. There also are numbers that may be too small for the Double data type to handle, such as the size of an atom. However, these circumstances are relatively rare.*

The Boolean data type has only two possible values: True and False. The Boolean data type would be a good choice to report whether you got into the class because there are only two alternatives: yes (True) and no (False).

NOTE *The Boolean data type is considered numeric because 0 (zero) is considered False and all nonzero numbers True.*

Text Data Types

The String and Char data types are used for text.

A *string* is simply one or more characters, usually enclosed in double quotes to indicate that a string is intended. The characters may be alpha (A–Z or a–z), numeric (0–9), or virtually any other character you can type from your keyboard. For example, the name “R2D2” is a string even though it includes the numeric character 2. The String data type would be a good choice for the teacher’s name, such as “Genghis Khent,” my students’ fond (?) nickname for me.

The Char data type represents a single character, also enclosed in double quotes, followed by an upper- or lowercase *c* to indicate that it is a character rather than a string. As with a string, the character may be alpha (A–Z or a–z), numeric (0–9), or virtually any other character you can type from your keyboard. The Char data type would be a good choice for the grade you hope to earn in the class, such as an “A”.

There are other data types, some of which will be mentioned in later chapters. However, these five data types, Integer, Double, String, Char, and Boolean, are the ones principally used.

Data Types of Visual Basic Properties

Take a look at the Properties window of the form in your project. The form has many different properties. These properties determine the form’s height and width, background color, caption, visibility, and so on. Visual Basic 2005 uses these properties when you start a project to determine the form’s size, background color, and so forth.

Each of these properties stores a particular value. The Height property stores a number that represents the height of the form. The Text property stores a string that represents the title displayed by the form. The Visible property stores a Boolean value that represents whether the form is visible (True) or hidden (False).

You can access the values of many properties when designing your application (design time) simply by viewing them in the Properties window. You also can access the values of many properties while your application is running through code (run time). In Chapter 2, we changed the Text property of the form at run time; in Chapter 3, we changed the Text property of labels at run time.

However, whether you are in design time or run time, the new value of a property must be of the correct data type. To confirm this, in the Properties window of the form, type **Jeff** next to the Height property, which you can access by expanding the Size property, as shown in Figure 4-1. Then press ENTER. A dialog box will display, as in Figure 4-2, warning you of an “invalid property value.”

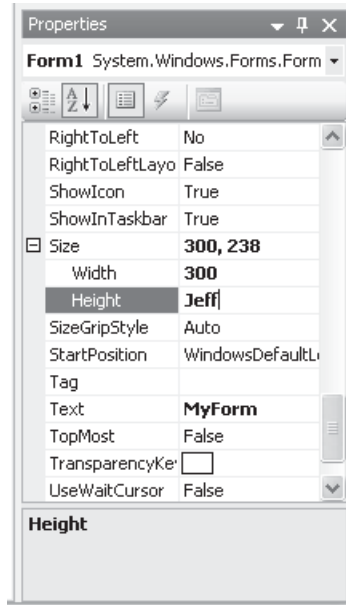


Figure 4-1 Setting the form’s Height property to an invalid value

Click the Details button of the dialog box in Figure 4-2. The dialog box then will display the message, “Jeff is not a valid value for Int32.” As discussed previously, System.Int32 is the name used in the .NET Framework for the Integer data type.

That Visual Basic 2005 prevents you from changing the value of the Height property to “Jeff” makes sense. The height must be a number. Visual Basic does not know how to make a form of the height “Jeff.”

Try exploring the properties of the form in the Properties window. You will see there are many different data types for the different properties.

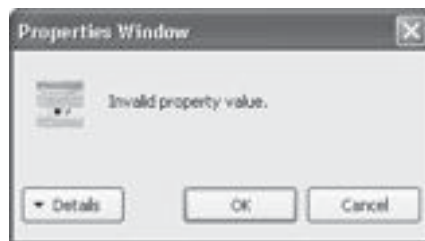


Figure 4-2 Invalid property value warning

Variables

You can store, access, and change the value of a property. However, you cannot change what the property stands for. For example, the Height property of a Form object represents the height of a form; you cannot change that property so that it instead represents the width of a form or the name of your favorite ice cream.

Instead, you can create a variable to store data of your choosing, such as the name of your favorite ice cream, your social security number, and so on.

Declaring a Variable

Visual Basic knows that the form's Height property stands for the height of the form and that its data type is numeric because the Height property is built into the .NET Framework class library. However, because you, not Visual Basic, create a variable, you need to tell Visual Basic information about the variable. You do so by declaring the variable.

You declare a variable with the following syntax:

```
[Access Specifier] [Variable Name] As [Data Type]
```

To make this syntax more understandable, here are two examples of declaring a variable:

```
Dim intScore As Integer  
Private strName As String
```

In the first example, Dim is the access specifier, intScore is the variable name, and Integer is the data type. In the second example, Private is the access specifier, strName is the variable name, and String is the data type.

There are other access specifiers, but Dim and Private will be the ones used in this chapter.

You can choose any of the data types discussed in the preceding section on data types, though logically, you should choose a data type that is appropriate for the purpose of the variable. For example, if the variable represents someone's name, you likely will choose String as the data type, whereas if the variable represents someone's age, you instead may choose the Integer data type.

Naming a Variable

Variables, like people, have names. These names are used to identify the variable to which you want to refer. There are only a few limitations on how you can name a variable:

- The variable name cannot begin with any character other than a letter of the alphabet (A–Z or a–z) or an underscore (_). Secret agents may be named 007, but not variables.
- The variable name cannot contain embedded spaces, such as My Variable, or punctuation marks other than the underscore character (_), such as a question mark (?), comma (,), period (.), backslash (\), forward slash (/), or a parenthesis.
- The variable name cannot be longer than 255 characters (not that you would want to create a variable name that long).
- The variable name cannot be the same as a keyword, such as Integer or String, because that would confuse the compiler. (Technically, you can put the keyword in brackets to use it as a variable name, but as your mother may have told you, just because you can do something doesn't mean you should do it.)
- The variable name cannot have the same name as the name of another variable of the same scope, because that also would confuse the compiler. Scope is discussed later in this chapter.

Besides these limitations, you can name a variable pretty much whatever you want. However, it is a good idea to give your variables names that are meaningful. If you name your variables var1, var2, var3, and so on, through var17, you may find it difficult to remember later the difference between var8 and var9. And if *you* find it difficult, imagine how difficult it would be for another programmer who has to make sense of your code.

In Chapter 3, I recommended you use a naming convention when naming controls. I similarly recommend that you use a naming convention when naming your variables. Analogous to Chapter 3, the naming convention I suggest is to name a variable with a prefix, usually all lowercase and consisting of three letters, that indicates its data type, followed by a word, first letter capitalized, that suggests its purpose.

Here are some suggested prefixes for data types:

Integer	int
String	str
Boolean	bln
Double	dbl

Here are some examples that use these prefixes:

- **intScore** An Integer variable representing a score, such as on a test
- **strName** A String variable representing a name, such as a person's name
- **blnResident** A Boolean variable representing whether or not someone is a resident
- **dblGPA** A Double variable representing a student's GPA

If you need more than one word to describe the variable's purpose, you should combine the words into one (because you cannot have embedded spaces) but capitalize the first letter of each word, such as `blnDidUserQuit`.

What Happens If You Don't Declare a Variable?

By default (and I would not change this default), Visual Basic 2005 requires you to declare a variable before you refer to it in code.

For example, in either a new or existing Windows application, type the following code in the Load event procedure of the form. This code, which attempts to assign 10 to `intVar` without previously declaring `intVar` as a variable, will not compile.

```
Private Sub Form1_Load(ByVal sender As Object, _  
    ByVal e As System.EventArgs) Handles MyBase.Load  
    intVar = 10  
End Sub
```

Instead, on the line `intVar = 10`, the compiler will complain, "Name 'intVar' is not declared."

Where Do You Declare a Variable?

You can declare a variable in one of two places: inside a procedure or at the top of the code module. Where you declare a variable affects its scope.

Local Variable

If you declare a variable inside a procedure, you can refer to that variable only in that procedure. Stated in programming parlance, the variable is a *local* variable, having scope only inside the procedure in which it was declared. The `Dim` access specifier generally is used for local variables.

Assume the code in the Load and Click event procedures of the form read as follows:

```
Private Sub Form1_Load(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    Dim intVar As Integer
End Sub

Private Sub Form1_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles MyBase.Click
    intVar = 10
End Sub
```

You will get a compile error “Name ‘intVar’ is not declared.” The line `intVar = 10` inside the Click event procedure will be highlighted. The reason is that `intVar` only has scope inside the Load event procedure in which it was declared and therefore is not visible in the Click event procedure.

By contrast, assigning 10 to `intVar` inside the Load event is okay because `intVar` was declared inside that event procedure. Try this by deleting the line of code in the Click event procedure of your form and then changing the code in the Load event procedure of your form so it reads as follows:

```
Private Sub Form1_Load(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    Dim intVar As Integer
    intVar = 10
End Sub
```

In this example, the variable `intVar` was declared in the first statement and assigned a value in the second statement. You also can combine the two statements as follows:

```
Dim intVar As Integer = 10
```

Combining the declaration and assignment of a variable within one statement is called *initialization*.

Module-Level Variable

You also can declare a variable outside of and above any procedure. In programming parlance, this is a *module-level* variable, having scope in all event procedures and other code in that code module. Either the `Dim` or `Private` access specifier may be used for module-level variables; there is essentially no difference between them.

In the following example, because `intVar` is declared at the module level, it can be accessed by both the `Load` and `Click` event procedures of the form, without having been declared inside those event procedures:

```
Private intVar As Integer
Private Sub Form1_Load(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    intVar = 10
End Sub

Private Sub Form1_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles MyBase.Click
    intVar = 5
End Sub
```

You may declare a module-level variable using initialization:

```
Private intVar As Integer = 10
```

However, you can assign a value to an already declared variable only inside a procedure; you cannot do so at the top of the code module:

```
Private intVar As Integer
intVar = 10
```

Instead, you will get the compile error “Declaration expected.”

Why Not Always Declare Variables at the Module Level?

Given the potential for compiler errors resulting from variables being referenced outside their scope, the temptation is to give your variables the widest possible scope and make them module level instead of local. Resist temptation! Indeed, as a general rule, you should make your variables local rather than module level, giving them the least amount of scope possible.

One reason is, when debugging your code, if a variable can be accessed only from one location in your program, you only need to check the code in that one place. However, if the variable can be accessed from ten different locations in your program, you need to check the code in all ten places, as well as determine the effect of any interrelationships between the ten locations. In other words, the less scope the variable has, the easier your task as a programmer. Why make your job harder than it has to be?

Of course, there will be circumstances in which a variable should have module-level scope. The point is that, in determining whether to declare a variable locally in an event procedure or instead at the module level, you should not declare the variable at the module level unless you can justify to yourself why you need to do so.

Constants

A constant is similar to a variable, except that a constant's value cannot change during the life of the program.

Declaring a Constant

The syntax of declaring a constant is similar to a variable:

```
Const [Constant Name] As [Data type] = [value]
```

For example, the following statement declares a constant, `MAX_SCORE`, of the data type `Integer`, whose value, `100`, is the maximum score that can be obtained on a test:

```
Const MAX_SCORE As Integer = 100
```

Let's analyze the component parts of the constant declaration:

- **Const** `Const` is a keyword that indicates you are declaring a constant instead of a variable.
- **Constant name** The naming convention for constants is different than for variables. By convention, constant names, unlike variable names, do not have a prefix such as `int` or `str` to specify the data type, but instead are entirely descriptive. Additionally, by convention the name consists of uppercase characters, so words are separated by an underscore character (`_`), as in `BRIBE_PAID`.
- **Data type** Same as with variables.
- **Assigning a value** The main difference in syntax between declaring a variable and declaring a constant is that a constant must be assigned a value when declared. The reason why a constant must be assigned a value when it is declared is that the value of a constant cannot be changed after it is declared. Therefore, a constant must be given a value when it is declared or it can never be given a value at all.

Where Do You Declare a Constant?

You can declare a constant at either local scope (inside an event procedure) or module-level scope (outside and above the event procedures). The reasons why I recommend you declare a variable locally, unless you have a specific reason to declare the variable at the module level, don't apply to constants because, as the next section shows, you can't change the value of a constant after you declare it.

Usually constants are declared at the module level, so they can be used by all of the form's event procedures. An access specifier may but need not be used when the constant is declared at the module level.

Where Do You Assign a Value to a Constant?

The answer is that you only can assign a value to a constant when you declare it (that is, initialization).

Because a constant's value cannot be changed during the life of the program, even attempting to assign a value to a constant will cause an error. Try this code in the Windows application you have been using in this chapter:

```
Const MAX_SCORE As Integer = 100
Private Sub Form1_Load(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    MAX_SCORE = 200      'error
End Sub
```

Why Use Constants?

Although it is important to know how a constant differs from a variable and how to declare constants, you may be wondering, Why use constants at all? The reason is that constants make your code easier to read and maintain.

Although constants are useful for values that never will change, constants perhaps are even more useful for values that someday may change. For example, we've all paid sales tax on purchases. Assuming the tax rate is 8%, the amount of the tax is price * .08. Thus, throughout your code for a store you may have calculations such as the following:

```
[price variable] * .08
```

One day the government decides to increase the sales tax to 8.25%. Now you have to find all the places in your code where you referred to the sales tax rate and change all those references from .08 to .0825. This not only is a pain, but the potential for error is obvious.

Alternatively, you could have declared the sales tax rate as a constant:

```
Const SALES_TAX_RATE As Double = .08
```

Thus, the tax calculation in your code would be as follows:

```
[price variable] * SALES_TAX_RATE
```


Then, when the government increases the sales tax to 8.25%, you only have to make the change in one place in your code, and you're done:

```
Const SALES_TAX_RATE As Double = .0825
```

Conclusion

Most programs need to keep track of information. That information may be about the subject of the program, such as the names and addresses of customers, or it may be about the program itself, such as the height, caption, or visibility of a form.

Data comes in different forms. Data may be numeric (such as the height of a form), text (such as the caption on a form), or Boolean (such as whether a form is visible). The type of information, whether number, string, or Boolean, is referred to as the *data type*.

Although the .NET Framework class library has many built-in properties to store data, Visual Basic 2005 also enables you to create your own information storage locations, called *variables*. Variables must be declared before they are used.

Variables may be declared at the top of the code module, in which case they are called *module-level variables*, and will be available to all procedures in that module. Variables also may be declared inside a procedure, in which case they are called *local variables* and their scope is limited to the procedure in which they were declared.

Finally, certain values never change during the life of the program. These unchanging values are represented by constants, which are declared similarly to variables. However, unlike variables, constants must be initialized when they are declared, and their value cannot thereafter change during the lifetime of the program.

In this chapter, you used the assignment operator to provide values to variables. In the next chapter, you will learn about arithmetic operators, which enable you to use the computer's unparalleled ability to quickly and accurately perform mathematical calculations.

Quiz

1. What does a data type signify?
2. What is a floating-point number?

3. Can you change the data type of a built-in property of a form, such as Height or Text?
4. What is the purpose of a variable?
5. Does Visual Basic 2005 by default require you to declare a variable before you refer to it in code?
6. What is a local variable?
7. What is a module-level variable?
8. Do you have to assign a value to a variable when you declare it?
9. What is the difference between a constant and a variable?
10. Do you have to assign a value to a constant when you declare it?

Letting the Program Do the Math—Arithmetic Operators

It is only fair that since my students have to listen to my recycled jokes, you have to read my recycled introductions. Back in Chapter 2 I complained that nowadays students don't need to be able to calculate arithmetic in their heads because they can rely on calculators. However, despite my complaining about calculators, they certainly are far faster and more accurate than I could ever hope to be. The reason is that a calculator is a computer, and computers are superstars when it comes to calculating.

You harness the computer's calculating ability using arithmetic operators. You will learn in this chapter how to enable your applications to make fast and accurate calculations using arithmetic operators. At the end of this chapter, you will put what you learned into practice with the Change Machine project, a type of calculator that converts a number of pennies into dollars, quarters, dimes, nickels, and pennies.

Arithmetic Operators

Visual Basic 2005 can do your arithmetic, and because a computer is involved, it's much faster and more accurate than any human! Even better, the code is relatively easy to write, because the syntax for arithmetic is quite similar to how you would write the arithmetic calculation on paper or how you would use a calculator.

Table 5-1 lists the arithmetic operators.

The Addition Operator

The addition operator works exactly as you would expect it to with numeric values. In the following code snippet, the third line of code adds the values of variables *a* and *b* and then assigns the sum, 5, to variable *a*, changing its value from 2 to 5:

```
Dim a As Integer = 2
Dim b As Integer = 3
a = a + b
```

Operator	Name	What It Does
+	Addition	Performs addition.
-	Subtraction	Performs subtraction.
*	Multiplication	Performs multiplication.
^	Exponentiation	Raises a number to a specified power.
/	Floating-point division	Performs floating-point division; the remainder is preserved and expressed as a decimal.
\	Integer division	Performs integer division; the remainder is dropped.
Mod	Modulus division	Used to obtain the remainder from division.

Table 5-1 Arithmetic Operators

The addition operator also works with String variables by concatenating, or appending, one string to another. In the following code snippet, the third line of code adds the values of variables `a` and `b` and assigns the concatenated string, “JeffKent”, to variable `a`, changing its value from “Jeff” to “JeffKent”:

```
Dim a As String = "Jeff"  
Dim b As String = "Kent"  
a = a + b
```

NOTE The `&` operator also performs the same function as the addition operator with strings.

The Subtraction Operator

The subtraction operator also works exactly as you would expect it to with numeric values. In the following code snippet, the third line of code subtracts the value of variable `b` from variable `a` and assigns the difference, `-1`, to variable `a`, changing its value from `2` to `-1`:

```
Dim a As Integer = 2  
Dim b As Integer = 3  
a = a - b
```

The Multiplication Operator

The multiplication operator also works exactly as you would expect it to with numeric values. In the following code snippet, the third line of code multiplies the value of variable `a` by the value of variable `b` and assigns the product to variable `a`, changing its value from `2` to `6`:

```
Dim a As Integer = 2  
Dim b As Integer = 3  
a = a * b
```

The Exponent Operator

The exponent operator (`^`) may not be as familiar, but its use is simple: 3 squared is expressed as `3 ^ 2`. In the following code snippet, the third line of code raises the

value of variable a (3) to the second power (the value of variable b) and then assigns the result (9) to variable a, changing its value from 3 to 9:

```
Dim a As Integer = 3
Dim b As Integer = 2
a = a ^ b
```

The Division Operators

Whereas there is only one addition, subtraction, and multiplication operator, there are three division operators. The operators /, \, and Mod all involve division. However, one important difference among the three division operators is how they report the results of the division.

Using as an example 11 divided by 4, the result is 2 remainder 3. In this example, 2 is the quotient and 3 is the remainder. The results reported by the three division operators are as follows:

- The / operator reports the entire result, 2 remainder 3, expressed as a decimal, 2.75.
- The \ operator reports only the quotient, 2, and drops the remainder. Integer division does not round off. If it did, 11 \ 4 would be 3, not 2. Because integer division reports only the quotient, the result necessarily is a whole number.
- The Mod operator reports only the remainder, 3, and drops the quotient. Because modulus division reports only the remainder, the result necessarily is a whole number.

You often will use the / operator, which performs floating-point division, because it provides you the complete result. However, the Change Machine project at the end of this chapter shows you that the \ and Mod operators also can be very useful.

Tip *Programmers sometimes find it difficult to recall which of the / and \ operators is floating-point division and which is integer division. One mnemonic is that the / is a forward slash, and the f in “forward” corresponds to the f in “floating point.” Another memory technique is that the / looks more like the normal arithmetic division operator than does the \, and floating-point division produces the normal quotient and remainder result of arithmetic division, whereas integer division does not.*

Operator Precedence

So far the arithmetic expressions have been simple, involving just one arithmetic operator. However, sometimes arithmetic expressions are more complex, involving two or more arithmetic operators. For example, does the arithmetic expression $2 + 3 * 4$ equal 20 (by performing addition before multiplication) or 14 (by performing multiplication before addition)?

One and only one of these two answers can be correct. Rules of operator precedence are necessary to determine which of the two answers is correct.

Table 5-2 lists the order of precedence, or priority, among arithmetic operators.

Thus, $2 + 3 * 4$ equals 14, because multiplication has a higher priority than addition and therefore is performed first.

Because multiplication and division have equal priority, when both operators occur together in an expression, priority goes from left to right. Therefore, whichever of the two operators is on the left is performed before the one on the right. The same left-to-right priority rule applies between addition and subtraction.

Parentheses can be used to override the order of precedence and force some parts of an expression to be evaluated before others. Operations within parentheses are always performed before those outside the parentheses. Thus, $(2 + 3) * 4$ equals 20, not 14, because the parentheses force addition to be performed first.

Combining Arithmetic and Assignment Operators

As discussed earlier in this chapter, in the following code snippet, the third line of code adds the values of variables *a* and *b* and assigns the sum, 5, to variable *a*, changing its value from 2 to 5:

```
Dim a As Integer = 2
Dim b As Integer = 3
a = a + b
```

Priority	Operator(s)	Description
1	^	Exponent
2	-	Unary negation operator (not subtraction)
3	*, /	Multiplication and floating-point division
4	\	Integer division
5	Mod	Modulus (remainder)
6	+, -	Addition and subtraction, string concatenation

Table 5-2 Operator Precedence

A precedence issue arises here in the third line of code. Even though there is only one arithmetic operator, there are two operators, one arithmetic and the other assignment. However, the precedence issue is easily resolved. Addition is performed before assignment because all arithmetic operators have precedence over the assignment operator.

The third statement can be shortened as follows and still accomplish the same result:

```
a += b
```

The combined arithmetic/assignment operators are shown in Table 5-3.

These shorthand arithmetic/assignment operators make your code more readable. The purpose of the following statement is to increment (increase by 1) the value of variable a:

```
a += 1
```

The purpose of that statement is more readable (as well as shorter to type) than the following statement:

```
a = a + 1
```

The Mod operator has no corresponding arithmetic/assignment operator because the remainder of a variable divided by itself is always 0.

The Parse and ToString Methods

As discussed earlier in this chapter, the addition operator works with String values as well as with numeric values. With String values, the addition operator concatenates, or appends, one string to another.

The ability of the addition operator to perform double duty with String as well as numeric values can backfire on you. To illustrate, assume your application has two

Operator	Use	Alternate
+=	a += b	a = a + b
-=	a -= b	a = a - b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
\=	a \= b	a = a \ b
^=	a ^= b	a = a ^ b

Table 5-3 Combined Arithmetic/Assignment Operators

TextBox controls, txtOp1 and txtOp2, in which the user types two numbers to be added, with the sum displayed in a Label control named lblResult. The application may use the following code:

```
lblResult.Text = txtOp1.Text + txtOp2.Text
```

The user wants to add $2 + 2$, so types 2 in each TextBox. However, the answer is not the expected 4, but instead 22! This is not new math. Instead, Visual Basic assumed you intended to concatenate two strings (“2” + “2” = “22”) instead of adding two numbers ($2 + 2 = 4$) because the data type of the Text property of the two TextBox controls is a String, not a numeric data type.

The solution is to explicitly direct, through code, that Visual Basic convert the string representation of an integer (the Text property of txtOp1 and txtOp2) into actual integer values before performing addition and then assigning that sum to be displayed in lblResult. You can accomplish this conversion through the Parse method of the Integer class. This method converts its argument, the string representation of an integer, into an actual integer value before that value is assigned to the Integer variable. The following code in the program converts the string representations of each of the two integers (the Text properties of txtOp1 and txtOp2, respectively) to the actual integer values before adding those values and assigning the resulting sum to be displayed in lblResult:

```
lblResult.Text = Integer.Parse(txtOp1.Text) + _  
    Integer.Parse(txtOp2.Text)
```

NOTE The Double class also has a Parse method, which converts the string representation of a floating-point number into an actual number (“123.45” into 123.45).

The ToString method is the converse of the Parse method. Whereas the Parse method converts a string representation of a number into a number (“123” into 123), the ToString method converts a number into the string representation of a number (123 into “123”). This can be useful when you want a number displayed in a control whose Text property is a string. The following code first stores the sum in the Integer variable sum and then uses the ToString method to convert that integer into the string representation of an integer before assigning it to the label’s Text property, whose data type is a String:

```
Dim sum As Integer  
sum = Integer.Parse(txtOp1.Text) + _  
    Integer.Parse(txtOp2.Text)  
lblResult.Text = sum.ToString
```

All classes have a ToString method. What that method does depends on the class. In the case of the Int32 class, which represents an integer, the ToString method converts an integer to the string representation of the integer so it can be assigned to the Text property of the Label controls.

The ToString method, as used with the Integer class, is preceded by the integer value to be converted and then a dot (or period). It is followed by empty parentheses because this method has no parameters.

We will be using the Parse and ToString methods in the Change Machine project later in this chapter.

Class Methods

In previous chapters, we have discussed how classes have properties and events. A *property* is a characteristic of an object of a class, such as the Text property of the Button class being the text displayed on a button, such as “Calculate” or “Clear.” An *event* is something that happens to an object of a class, such as the Click event of the Button class being the event that occurs when a button is clicked.

Parse and ToString are not properties or events, but methods of a class, such as Integer. A *method* is something an object of a class does. For example, as objects of the Person class, our methods could include breathe, walk, talk, and so on. The Form class (among others) also has methods, as you will learn in later chapters.

Change Machine Project

My mother was not above using a change machine to distract cranky or mischievous young grandchildren. The youngsters poured hundreds of pennies into the top of the machine and watched with fascination (fortunately youngsters are easily fascinated) as the machine sorted the pennies into amounts of change that could be taken to the bank and exchanged for dollars, quarters, and so on. The youngsters were motivated as well as fascinated, because guess who got to keep the quarters?

Your project will ask the user to input the number of pennies. You can assume the user will input a positive whole number and then click the Calculate button. The code then will output in controls the number of dollars, quarters, dimes, nickels and pennies. Figure 5-1 shows the result of running the program and inputting 392 for the number of pennies.



Figure 5-1 Change Machine project in action

Creating the Project

Implement the following steps to create the project:

1. Start a new Windows application. I called my project name Change Machine.
2. Using the Toolbox, add controls to the form so that it appears as shown in Figure 5-2. All the controls are labels except for the two buttons on the bottom of the form and the text box across from the label caption “Enter Pennies.”
3. Using the Properties window, change the Name property of the TextBox control to txtPennies and then delete any value in its Text property.
4. Using the Properties window, change the AutoSize property of all labels from the default True to False. This can be done by selecting all the labels first, which changes the AutoSize property of each. This step will make easier the customization of the labels in the following steps.
5. Using the Properties window, change the Text properties of the labels on the left so they are captions as they appear in Figure 5-2.
6. Using the Properties window, change the Name properties of the labels on the right to lblDollars, lblQuarters, lblDimes, lblNickels, and lblPennies, respectively.

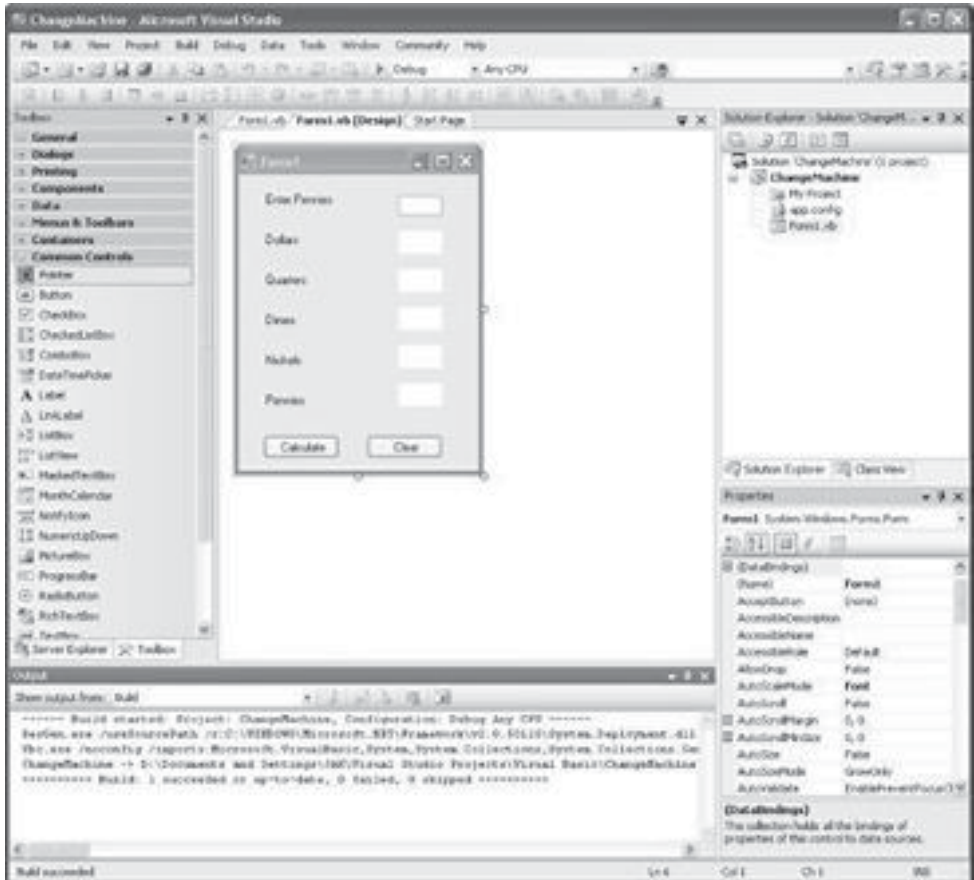


Figure 5-2 Form in design mode

7. Again using the Properties window, change the BackColor property of the labels on the right to White (so they will be more visible after we delete their text in the next step). When you click the value of the BackColor property, a tabbed dialog box appears. Choose the Custom tab and then click on a box that is white.
8. Also using the Properties window, delete any value in the Text properties of the labels on the right so they are blank, to avoid these labels' names displaying as the labels' text when the project first starts up.

- Using the Properties window, change the Name property of the button on the left to `btnCalculate` and its Text property to Calculate. Similarly, change the Name property of the button on the right to `btnClear` and its Text property to Clear.
- Create an event procedure stub for the Click event of `btnCalculate` and write the following code (to be explained in the following section, “The Algorithm”) inside the event procedure:

```
Private Sub btnCalculate_Click(ByVal sender As Object, _  
ByVal e As System.EventArgs) Handles btnCalculate.Click  
    Dim intLeftover As Integer  
    intLeftover = Integer.Parse(txtPennies.Text)  
    lblDollars.Text = (intLeftover \ 100).ToString  
    intLeftover = intLeftover Mod 100  
    lblQuarters.Text = (intLeftover \ 25).ToString  
    intLeftover = intLeftover Mod 25  
    lblDimes.Text = (intLeftover \ 10).ToString  
    intLeftover = intLeftover Mod 10  
    lblNickels.Text = (intLeftover \ 5).ToString  
    intLeftover = intLeftover Mod 5  
    lblPennies.Text = intLeftover.ToString  
End Sub
```

- Create an event procedure stub for the Click event of `btnClear` and write the following code inside the event procedure:

```
Private Sub btnClear_Click(ByVal sender As Object, _  
ByVal e As System.EventArgs) Handles btnClear.Click  
    txtPennies.Text = ""  
    lblDollars.Text = ""  
    lblQuarters.Text = ""  
    lblDimes.Text = ""  
    lblNickels.Text = ""  
    lblPennies.Text = ""  
End Sub
```

This code simply resets the Text properties of the TextBox and the Label controls on the right to blank, as they were when the application first started. The result is shown in Figure 5-3.

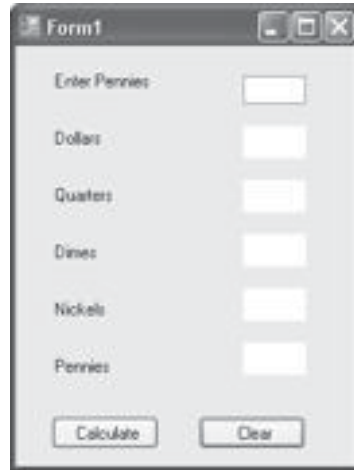
The image shows a screenshot of a Windows application window titled "Form1". Inside the window, there are six text input fields arranged vertically, each with a label to its left: "Enter Pennies", "Dollars", "Quarters", "Dimes", "Nickels", and "Pennies". At the bottom of the form, there are two buttons: "Calculate" on the left and "Clear" on the right. The "Clear" button has been clicked, as indicated by the caption.

Figure 5-3 Form at run time after the Clear button has been clicked

The Algorithm

As you learned in Chapter 1, the purpose of Visual Basic 2005, and indeed programming languages generally, is to enable you, as the programmer, to give instructions to the computer to carry out. Before you can formulate those instructions in code, you first need to be able to articulate those instructions in English or whatever other language you think in.

To write the Change Machine project, you need to come up with a step-by-step logical procedure to convert the pile of pennies into neater stacks of dollars, quarters, dimes, nickels, and pennies. A step-by-step logical procedure for solving a problem is called an algorithm, pronounced “Al Gore rhythm.”

One algorithm for converting the pile of pennies into dollars, quarters, dimes, nickels, and pennies is to first determine how many stacks of one hundred pennies you can make from the pile. Each stack of one hundred pennies would then represent one dollar. You then would work with the number of pennies left over to determine the number of quarters, dimes, nickels, and, finally, pennies.

For example, assume there are 392 pennies in the pile. You might use the following steps to determine the number of dollars, quarters, dimes, nickels, and pennies in 392 pennies:

- There are 100 pennies in a dollar. You can make three stacks of 100 pennies from 392 pennies. That means there are three dollars, with 92 pennies left over, from which you will determine the number of quarters, dimes, nickels, and pennies.

- There are 25 pennies in a quarter. You can make three stacks of 25 pennies from 92 pennies. That means there are three quarters, with 17 pennies left over, from which you will determine the number of dimes, nickels, and pennies.
- There are ten pennies in a dime. You can make one stack of ten pennies from 17 pennies. That means there is one dime, with seven pennies left over, from which you will determine the number of nickels and pennies.
- There are five pennies in a nickel. You can make one stack of five pennies from seven pennies. That means there is one nickel, with two pennies left over, which is the number of pennies.

Let's now convert this algorithm from English to code.

The first step is to store the number of pennies entered by the user in the TextBox control `txtPennies` into the Integer variable `intLeftover`. The following code does this, first using the `Parse` method of the Integer class (discussed in the earlier section “The Parse and ToString Methods”) to convert the string representation of an integer (the `Text` property of `txtPennies`) to the actual integer value before that value is assigned to the Integer variable (`intLeftover`):

```
intLeftover = Integer.Parse(txtPennies.Text)
```

When you divide the number of pennies (stored in `intLeftover`) by 100 (the number of pennies in a dollar), the quotient is the number of dollars in the pennies, and the remainder is the number of pennies left over. Integer division provides you with the quotient but no remainder, and the Mod operator provides you with the remainder:

```
lblDollars.Text = (intLeftover \ 100).ToString  
intLeftover = intLeftover Mod 100
```

NOTE As explained in the earlier section “The Parse and ToString Methods,” the `ToString` method converts a number (`intLeftover \ 100`) into the string representation of that number so it can be displayed as text in the Label control.

The quotient, representing the number of dollars in the pile of pennies, is displayed in `lblDollars`. The remainder is stored in `intLeftover`, which will be used in the code to determine the number of quarters, dimes, nickels, and pennies.

Next, you follow the same procedure, with two differences. First, you are not dividing the total number of pennies, but instead the number of pennies left over, represented by the current value of the variable `intLeftover`. Second, you are not dividing by 100, but instead by 25, the number of pennies in a quarter. We already have determined the number of dollars in the pile of pennies. Now we want to

determine the number of quarters in the remaining pennies. Accordingly, the code reads as follows:

```
lblQuarters.Text = (intLeftover \ 25).ToString  
intLeftover = intLeftover Mod 25
```

The remainder of the code follows the same process, except that next the divisor is 10, the number of pennies in a dime, then 5, the number of pennies in a nickel:

```
lblDimes.Text = (intLeftover \ 10).ToString  
intLeftover = intLeftover Mod 10  
lblNickels.Text = (intLeftover \ 5).ToString  
intLeftover = intLeftover Mod 5
```

The number of pennies left over after division by 5 cannot be converted into any higher change, so there is no need for further division:

```
lblPennies.Text = intLeftover.ToString
```

You frequently will need to create and implement algorithms in writing a computer program. Creating algorithms is a skill that can be developed from any field that requires analytical thinking, including but not limited to mathematics and computer programming.

Type Conversions

The following code in the Change Machine project uses the Parse method of the Integer class to convert the string representation of an integer (the Text property of txtPennies) to the actual integer value before that value is assigned to the Integer variable (intLeftover):

```
intLeftover = Integer.Parse(txtPennies.Text)
```

Without using the Parse method of the Integer class, the code instead would have read as follows:

```
intLeftover = txtPennies.Text
```

This code tries to assign a String value to an Integer variable. This appears to look like the data type equivalent of trying to put a square peg in a round hole. Yet, it works. The reason is that Visual Basic is smart enough to understand what you are trying to do, and converts the string representation of an integer into an integer before assigning the value to an Integer variable.

Because this conversion is done by Visual Basic behind the scenes, it is referred to as an *implicit* conversion. By contrast, using the Parse method of the Integer class to explicitly (through your code) convert the string representation of an integer into an integer is referred to as an *explicit* conversion.

The problem with implicit conversion is that Visual Basic does not always correctly guess your intentions. An example is the $2 + 2 = 22$ result in the preceding section “The Parse and ToString Methods.” Accordingly, to be on the safe side, you can require conversions that have the potential for problems if done implicitly instead of explicitly. You set this requirement by setting Option Strict to On (as opposed to the default Off).

There are two ways of setting Option Strict to On. One way is the statement Option Strict On at the top of your code:

```
Option Strict On
Public Class Form1
'remainder of code
```

Option Strict also can be set to On through the Options dialog box, which is displayed by choosing Options from the Tools menu. Figure 5-4 shows the project defaults, which are displayed by, in the left pane, selecting Projects and Solutions and then VB Defaults. You change the setting for Option Strict from Off (the default) to On using the drop-down box.

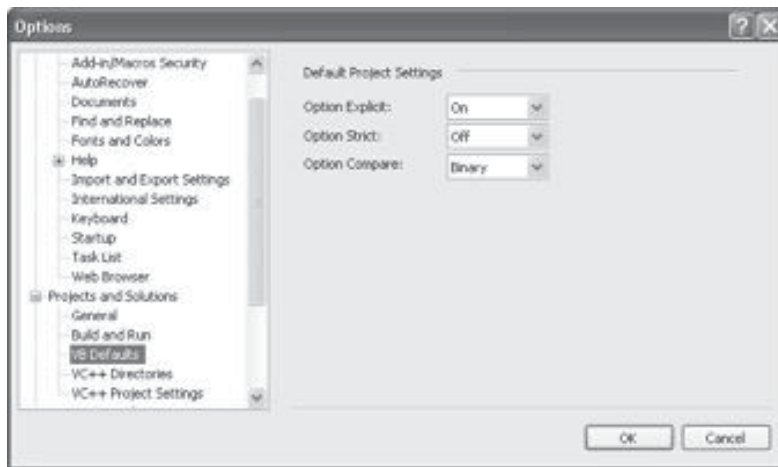


Figure 5-4 Project defaults

Once Option Strict is set to On, the following line of code will result in a compiler error:

```
intLeftover = txtPennies.Text
```

The compiler will give the following error message: “Option Strict On disallows implicit conversions from ‘String’ to ‘Integer.’”

Similarly, the following line in the Change Machine project uses the ToString method to convert the Integer result of an expression (`intLeftover \ 100`) to a String before assigning that result to a label’s Text property, whose data type is a string:

```
lblDollars.Text = (intLeftover \ 100).ToString
```

If instead you tried

```
lblDollars.Text = intLeftover \ 100
```

the result again would be a compiler error, the compiler giving you the following error message: “Option Strict On disallows implicit conversions from ‘Integer’ to ‘String.’”

You might legitimately be wondering what is so great about creating compiler errors that would not exist when Option Strict is Off. The answer is, it is much easier to fix a compiler error than it is to try to figure out the cause of a logic error such as $2 + 2 = 22$. An easy-to-fix compiler error is a small price to pay to avoid the headache of diagnosing a logic error.

Conclusion

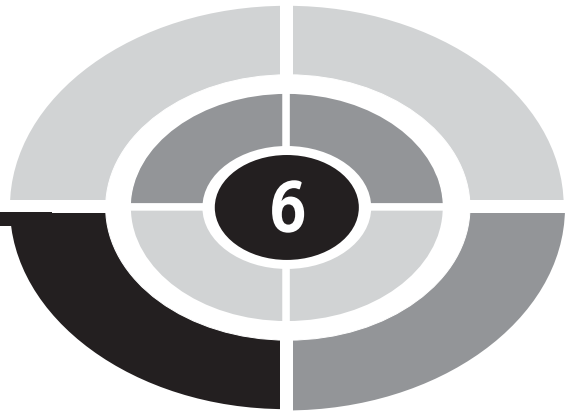
Computers, in addition to being able to store vast amounts of data, can calculate far faster and more accurately than we can. You harness the computer’s calculating ability using arithmetic operators. Most of the arithmetic operators, such as those for addition and multiplication, work the same as the arithmetic operators you have used with pencil and paper. The division operators include integer division (`\`), which returns just the quotient but not the remainder, and modulus division (`Mod`), which returns just the remainder and not the quotient.

In the next chapter, you will learn about relational and logical operators, which enable your program to take different actions depending on choices the user makes while the program is running.

Quiz

1. Which arithmetic operator works with string as well as numeric variables?
2. What is the significance of operator precedence?
3. How can you override default operator precedence?
4. Which operator raises a number to a specified power?
5. What is the difference between the / operator and the \ operator?
6. Which operator provides only the remainder resulting from division?
7. Which operator has precedence, an arithmetic operator or the assignment operator?
8. What is the purpose of the Parse method of the Integer class?
9. What is the purpose of the ToString method of the Integer class?
10. What is a method of a class?

This page intentionally left blank



CHAPTER

Making Comparisons— Comparison and Logical Operators

Can you imagine going to a restaurant that had only one item on its menu? Although this would make it easy for you to decide what you want to order, this one-item restaurant likely would not be in business long, because people like choices. Indeed, life is full of choices—some pleasant (a good menu) and some not so pleasant (do you want to pay by cash, check, or credit card?).

Up to now the programs we have discussed have been like the one-item restaurant, offering no choices. However, as programs become more sophisticated, they often branch in two or more directions. For example, in a calculator program, your program first would give the user a choice of whether they want to add, subtract, multiply, or divide. Your code then would need to determine which choice the user made before performing the indicated arithmetic operation, which will be different, and lead to a different result, depending on the user's choice. Your code would determine the user's choice by comparing it with the alternatives—addition, subtraction, multiplication, or division. You will learn in this chapter how to make such a comparison using comparison operators.

A comparison operator can make only one comparison at a time. Sometimes you need to combine several comparisons. For example, some years ago car washes had Ladies Free Wednesdays, which meant that on Wednesdays (evidently a slow day for car washes) women could have their cars washed for free. The car wash would need to make two comparisons to determine eligibility for a free car wash. The customer's gender must be equal to female, and the day of the week must be equal to Wednesday. Either comparison just by itself would not be enough to determine eligibility for a free car wash; the two comparisons must be done together. You will learn in this lesson how to combine several comparisons using logical operators.

The comparison and logical operators lay the groundwork for the following chapters on control structures and loops, which use these operators to determine if a condition, or a combination of conditions, evaluate as True or False.

Debugging

Before discussing the comparison and logical operators, let's take a brief detour into debugging. The immediate benefit of debugging is that it will enable you to test code in this chapter without going to the trouble of adding controls to your form. The longer-term benefit of debugging, which you will use in later chapters, is to enable you to identify and solve "bugs," a term that usually means a logic error in your code (such as $2 + 2 = 22$ instead of 4).

NOTE *The origin of the term "bug" is in dispute. One story is that during the pre-PC era, when mainframe computers ruled the earth, a mainframe was producing illogical results. The programmers checked and rechecked their punch cards but could find no errors. In desperation, they opened up the mainframe. Inside they saw a moth fried on one of the circuits.*

The WriteLine method of the Debug class is useful in debugging programs. The syntax of the WriteLine method is shown here:

```
Debug.WriteLine (parameter)
```

The WriteLine method outputs the value of the parameter to the Output window, which you can display with the menu command View | Other Windows | Output. For example, the following code outputs 10 to the Output window:

```
Private Sub Form1_Load(ByVal sender As Object, _  
    ByVal e As System.EventArgs) Handles Me.Load  
    Dim A As Integer = 10  
    Debug.WriteLine(A) ' Outputs 10  
End Sub
```

The WriteLine method only outputs to the Output window if you start your application with the Debug | Start menu command. There will be no output to the Output window if you instead start your application with the Debug | Start Without Debugging menu command. This is logical because you need to be debugging to use the Debug class.

Finally, the output to the Output window from the WriteLine method usually is not the only output in the Output window. The Output window normally also contains information generated by Visual Basic 2005. As Figure 6-1 shows, the output to the Output window from the WriteLine method usually is the last output in the Output window.

When finished debugging, choose Stop Debugging from the Debug menu.

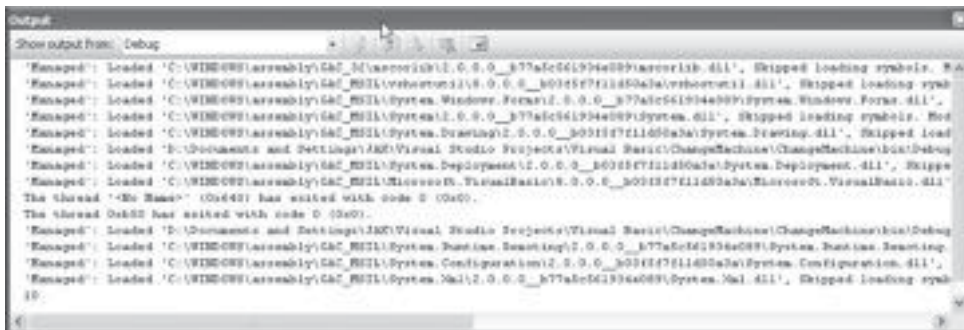


Figure 6-1 The Output window

Comparison Operators

Often your programs will need to compare two values. The comparison may be whether the two values are equal, or whether one value is greater (or less) than another. Regardless of which comparison is being made, the comparison may have only one of two possible results, either True or False.

Earlier we used the example of a calculator program to show one use of comparisons—to determine which of several choices the user made. Comparisons also are used for error prevention. For example, in the calculator program, before performing division, the program should compare the divisor to zero, because division by zero is illegal and, if performed, will result in a run-time error. If the divisor is equal to zero, the user should be warned and the division not performed. Otherwise, the division may be performed.

Comparison operators usually are used to compare numerical values, but some of them also may be used to compare strings, as discussed later in this chapter.

The syntax of a comparison is shown here:

```
[Expression1] [comparison operator] [Expression2]
```

In the following discussion, the term “left expression” refers to the expression on the left side of the comparison operator (Expression1 in the sample syntax). Similarly, the term “right expression” refers to the expression on the right side of the comparison operator (Expression2 in the sample syntax).

Both the left and right expressions may be anything that has a value that can be compared: literals, constants, variables, or properties. However, the data type of the two expressions should be the same.

Numeric Comparison Operators

The following paragraphs list and describe the comparison operators used to compare numbers and the circumstances under which they evaluate to True or False.

The less than operator (<) results in the expression being True if the left expression is less than the right expression, such as $4 < 5$, but False if the left expression is greater than or equal to the right expression, such as $5 < 4$ or $5 < 5$.

The less than or equal to operator (<=) results in the expression being True if the left expression is less than or equal to the right expression, such as $4 <= 5$ or $5 <= 5$, but False if the left expression is greater than the right expression, such as $5 <= 4$.

The greater than operator ($>$) results in the expression being True if the left expression is greater than the right expression, such as $5 > 4$, but False if the left expression is less than or equal to the right expression, such as $4 > 5$ or $5 > 5$.

The greater than or equal to operator ($>=$) results in the expression being True if the left expression is greater than or equal to the right expression, such as $5 >= 4$ or $5 >= 5$, but False if the left expression is less than the right expression, such as $4 >= 5$.

The equality operator ($=$) results in the expression being True if the left expression is equal to the right expression, such as $5 = 5$, but False if the left expression is less than or greater than the right expression, such as $4 = 5$ or $5 = 4$.

NOTE The equality comparison operator ($=$) is overloaded, also serving as an assignment operator. The compiler can tell whether you are using the $=$ operator for assignment or comparison based on the context in which the operator is used.

The inequality operator ($<>$) works the opposite of the equality operator. The inequality operator results in the expression being True if the left expression is less or greater than the right expression, such as $4 <> 5$ or $5 <> 4$, but False if the left expression is equal to the right expression, such as $5 <> 5$.

Try running the following code in a new or existing project. The output for each `Debug.WriteLine` statement, True or False, is in the comment accompanying that line:

```
Private Sub Form1_Load(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles Me.Load
    Dim A As Integer = 10
    Dim B As Integer = 8
    Dim C As Integer = 10
    Debug.WriteLine A > B ` Outputs True
    Debug.WriteLine A >= B ` Outputs True
    Debug.WriteLine A = B ` Outputs False
    Debug.WriteLine A <> B ` Outputs True
    Debug.WriteLine A < B ` Outputs False
    Debug.WriteLine A <= B ` Outputs False
    Debug.WriteLine A > C ` Outputs False
    Debug.WriteLine A >= C ` Outputs True
    Debug.WriteLine A = C ` Outputs True
    Debug.WriteLine A <> C ` Outputs False
    Debug.WriteLine A < C ` Outputs False
    Debug.WriteLine A <= C ` Outputs True
End Sub
```

String Comparisons

Comparing two strings is quite similar to comparing two numbers. For example, “Jeff” = “Jeff” evaluates to True, whereas “Jeff” = “Kent” evaluates to False.

Programs often need to make string comparisons. For example, code that authenticates users who are logging in needs to compare the user name entered with a list of user names, and the password entered with the password for that user name. Another example is the Find feature in Microsoft Word, Internet Explorer, and other applications, which enables you to search text for specific words.

String comparisons are based on positive integer values of the characters in the string. For the English language, the character set adopted by ANSI (American National Standards Institute) and ASCII (American Standards Committee for Information Interchange) uses the numbers 0–255 to cover all alphabetical characters (upper- and lowercase), digits and punctuation marks, and even characters used in graphics and line drawings. Table 6-1 lists the ASCII values of commonly used characters.

The result of the comparison of string representations of numbers may not always be what you might expect. As you might expect, “5” is greater than “4” because the ASCII value of 5 (53) is greater than the ASCII value of 4 (52). However, “5” also is greater than “4444” for the same reason. In comparing two strings, if the value of the first character of one string is greater than the first character of the other, the values of the remaining characters of the two strings do not matter. Thus, the string “ZAAA” is greater than “AZZZ”. Only if the values of the first characters of the two strings are the same are the second characters of the two strings compared. The comparison will continue, character by character, until one of the following happens:

- A character of one string is different from the character in the same position (such as second, third, fourth, and so on) of the other string. Thus, “Jeffrey” is larger than “Jeffery” because the fifth character of the first string, “r,” has a higher ASCII value than the fifth character of the second string, “e.”
- One string runs out of characters (that is, the two strings are of different length), in which case the longer string is the greater. Thus, “Jeffrey” is larger than “Jeff” because the second string (“Jeff”) runs out of characters before the first string (“Jeffrey”).
- Both strings run out of characters at the same time (that is, the two strings are of equal length), in which case the two strings are equal. Thus, “Jeffrey” and “Jeffrey” are equal.

Characters	Values	Comments
0 through 9	48–57	0 is 48; 9 is 57.
A through Z	65–90	A is 65; Z is 90.
a through z	97–122	a is 97; z is 122.

Table 6-1 ASCII Values of Commonly Used Characters

Option Compare

As discussed previously, the ASCII values of lowercase alphabetical characters are greater than their uppercase counterparts—in other words, a is greater than A. The default in Visual Basic is that string comparisons are case sensitive—that is, they distinguish whether a character is uppercase or lowercase. Consequently, the string “jeff” is greater than, rather than equal to, “Jeff”.

Depending on the context of your program, you may want to make case-insensitive comparisons; that is, comparisons in which whether a character is uppercase or lowercase is irrelevant. In validating a user who is attempting to log on, for example, user names often are not case sensitive, whereas passwords often are.

You use the Option Compare statement to declare the default comparison method to use when string data is compared. The Option Compare statement may be one, but only one, of the following:

- Option Compare Binary
- Option Compare Text

Option Compare Binary is the default and is a case-sensitive comparison. Option Compare Text is a case-insensitive comparison. Thus, under Option Compare Binary, aaa is greater than AAA. However, under Option Compare Text, aaa is equal to AAA.

You can change the default Option Compare setting by using the Options dialog box shown in Figure 6-2. This dialog, which is displayed via the Tools | Options menu command, also was discussed in Chapter 5 in connection with the Option Strict statement.

You can change the Option Compare setting in one of two ways. One way is to choose Text from the drop-down box in the Options dialog box. The alternative is to declare an Option Compare Text statement above the beginning of the class declaration:

```
Option Compare Text
Public Class Form1
```

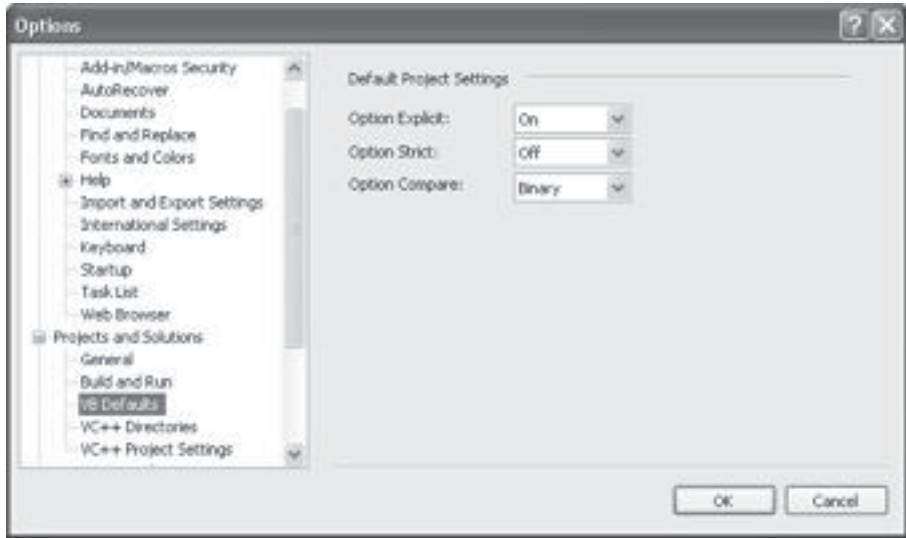


Figure 6-2 The Options dialog box

Like Operator

The Like operator is different from the preceding comparison operators in that it is used with strings rather than numbers, returning True if a string matches a specified pattern, False if it does not. Here is the syntax for the Like operator:

```
[string] Like [pattern]
```

Pattern matching often is used in everyday computing activities. For example, in searching for a file on your computer that you know starts with “msado” and has the extension .dll, you could do a search for the file msado*.dll, using the wildcard character *.

Table 6-2 lists some of the pattern-matching characters.

Characters in Pattern	Matches in String
?	Any single character
*	Zero or more characters
#	Any single digit (0–9)

Table 6-2 Pattern-Matching Characters

The wildcard character `*` is commonly used for searches, particularly when not all of the details of the string being searched for are known or remembered. The following comparison is True because “aBBBa” has an “a” at the beginning, an “a” at the end, and any number of characters in between:

```
"aBBBa" Like "a*a"
```

The wildcard character `?` provides for a more focused and therefore faster search than `*` because, whereas the `*` wildcard character can represent zero or more characters, the `?` wildcard character represents one character, no more and no less. The following comparison is True because “BAT” starts with a “B,” ends with a “T,” and has exactly one character in between:

```
"BAT" Like "B?T"
```

The wildcard characters `?` and `*`, like others, can be combined. The following comparison is True because “BAT” starts with a “B,” followed by any single character, followed by a “T,” and finally zero or more characters of any type:

```
"BAT123khg" Like "B?T*"
```

The wildcard character `#` provides for an even more focused search than the wildcard character `?` because, whereas the `?` wildcard character can represent any one character, the `#` wildcard character only can represent a character that is a digit. The following comparison is True because “a2a” begins and ends with an “a” and has exactly a single digit number in between:

```
"a2a" Like "a#a"
```

Precedence

Comparison operators rank lower than the arithmetic operators discussed in the previous chapter and higher than the logical operators discussed in the next section. All comparison operators are of equal precedence and are evaluated from left to right.

Logical Operators

Sometimes a first comparison and a second comparison both must evaluate as True for an action to take place. For example, a person may vote only if their age is at least 18 and they are a citizen:

- First comparison: age \geq 18
- Second comparison: USA citizenship = True
- Only if both comparisons are true: Allowed to vote
- If either comparison is false: Not allowed to vote

By contrast, at other times it is sufficient if either a first comparison or a second comparison evaluates as True for an action to take place. For example, to be admitted to a community college, the prospective student must be either at least 18 years old or have a high school diploma:

- First comparison: age \geq 18
- Second comparison: High school diploma = True
- If either comparison is true: Eligible for admission
- Only if both comparisons are false: Not eligible for admission

The combining of comparisons in either the conjunctive (and) or disjunctive (or) involves logical operators named, not surprisingly, And and Or, respectively. The following sections discuss these and other logical operators.

And Operator

As Table 6-3 shows, the And operator returns False unless both comparisons are True.

If the First Expression Is	And the Second Expression Is	The Result Is
True	True	True
True	False	False
False	True	False
False	False	False

Table 6-3 The And Operator

The following code shows the And operator in action:

```
Private Sub Form1_Load(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles Me.Load
    Dim A As Integer = 10
    Dim B As Integer = 8
    Dim C As Integer = 6
    Debug.WriteLine A > B And B > C      ' Outputs True
    Debug.WriteLine A > B And C > B      ' Outputs False
    Debug.WriteLine B > A And B > C      ' Outputs False
End Sub
```

In the first use of the And operator, $A > B$ And $B > C$, $10 > 8$ is True, and $8 > 6$ is True. Because both expressions are True, the output is True.

By contrast, in the second use of the And operator, $A > B$ And $C > B$, although $10 > 8$ is True, $6 > 8$ is False. Because only one expression is True and the other is False, the output is False.

Similarly, in the third use of the And operator, $B > A$ And $B > C$, $8 > 10$ is False, so even though $8 > 6$ is True, because one of the two expressions is False, the output is False.

Of course, if both expressions are False, the output is False.

The voting eligibility example discussed at the beginning of this section is a good example of when you would use the And operator, because both conditions (adult age and citizenship) must be True or the result (eligibility to vote) is False.

AndAlso Operator

The AndAlso operator is almost identical to the And operator in comparing two Boolean expressions. As Table 6-4 shows, the only difference is that if the first expression is False, the second expression is not evaluated.

The section “Why AndAlso and OrElse in Addition to And and Or?” later in this chapter discusses the consequences of the second expression not being evaluated and why you might use the AndAlso operator instead of the And operator.

If the First Expression Is	And the Second Expression Is	The Result Is
True	True	True
True	False	False
False	(not evaluated)	False

Table 6-4 The AndAlso Operator

Or Operator

As Table 6-5 shows, the Or operator returns True unless both comparisons are False.

The following code, which is the same as used for the And operator (except Or is substituted for And), shows the Or operator in action:

```
Private Sub Form1_Load(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles Me.Load
    Dim A As Integer = 10
    Dim B As Integer = 8
    Dim C As Integer = 6
    Debug.WriteLine A > B Or B > C      ' Outputs True
    Debug.WriteLine A > B Or C > B      ' Outputs True
    Debug.WriteLine B > A Or B > C      ' Outputs True
End Sub
```

In the first use of the Or operator, $A > B$ Or $B > C$, $10 > 8$ is True, and $8 > 6$ is True. Because both expressions are True, the output is True.

In the second use of the Or operator, $A > B$ Or $C > B$, $10 > 8$ is True, so even though $6 > 8$ is False, because at least one expression is True, the output is True.

Similarly, in the third use of the Or operator, $B > A$ Or $B > C$, whereas $8 > 10$ is False, $8 > 6$ is True, so again because at least one expression is True, the output is True.

Of course, if both expressions are False, the output is False.

The community college admission example discussed earlier is a good example of when you would use the Or operator, because only one of the two conditions (adult age or a high school diploma) needs to be True for the result (eligibility for admission) to be True.

The Or operator is implied in the comparison operators \geq and \leq . For example, the expression

```
A >= B
```

is the same as

```
A > B Or A = B
```

If the First Expression Is	And the Second Expression Is	The Result Is
True	True	True
True	False	True
False	True	True
False	False	False

Table 6-5 The Or Operator

OrElse Operator

The OrElse operator is to the Or operator what the AndAlso operator is to the And operator. As Table 6-6 shows, the only difference between the OrElse operator and the Or operator is that if the first expression is True, the second expression is not evaluated.

The section “Why AndAlso and OrElse in Addition to And and Or?” later in this chapter discusses the consequences of the second expression not being evaluated and why you might use the OrElse operator instead of the Or operator.

Xor Operator

The Xor operator performs a logical exclusion operation on two Boolean expressions and returns a Boolean value of True, as Table 6-7 shows, if one and only one of the expressions evaluates to True, and otherwise returns False.

The following code shows how the Xor operator works with Boolean expressions:

```
Private Sub Form1_Load(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles Me.Load
    Dim A As Integer = 10
    Dim B As Integer = 8
    Dim C As Integer = 6
    Debug.WriteLine A > B Xor B > C ' Outputs False
    Debug.WriteLine A > B Xor C > B ' Outputs True
    Debug.WriteLine B > A Xor B > C ' Outputs True
End Sub
```

In the first use of the Xor operator, $A > B \text{ Xor } B > C$, $10 > 8$ is True, and $8 > 6$ is True. Because both expressions are True, the output is False.

In the second use of the Xor operator, $A > B \text{ Xor } C > B$, $10 > 8$ is True, and $6 > 8$ is False. Because only one of the expressions is True, the output is True.

If the First Expression Is	And the Second Expression Is	The Result Is
True	(not evaluated)	True
False	True	True
False	False	False

Table 6-6 The OrElse Operator

If the First Expression Is	And the Second Expression Is	The Result Is
True	True	False
True	False	True
False	True	True
False	False	False

Table 6-7 The Xor Operator

Similarly, in the third use of the Xor operator, $B > A \text{ Xor } B > C$, whereas $8 > 10$ is False, $8 > 6$ is True. Because only one of the expressions is True, the output is True.

Of course, if both expressions are False, the output is False.

Not Operator

The Not operator changes True to False and False to True. An example is when my younger daughter tells me, “Dad, you look like Tom Cruise ... NOT!”

The Not operator is useful in situations in which Not True appears more intuitive than False. For example, in the calculator program discussed earlier, in verifying whether the divisor is equal to zero (division by zero being illegal), it may be more intuitive to say that division may be performed if the divisor is not equal to zero than to say that division may be performed if the divisor is greater than zero.

The Not operator is a unary operator, which means it operates on one operand. This is different from the preceding operators, which are binary, operating on two operands.

Precedence

Logical operators rank lower than the comparison operators discussed earlier in this chapter. Table 6-8 lists the order of precedence among comparison operators, from highest to lowest.

Priority	Operator(s)	Description
1	Not	Negation
2	And, AndAlso	Conjunction
3	Or, OrElse	Disjunction
4	Xor	Exclusion

Table 6-8 Precedence among Logical Operators

If the logical operators of equal priority appear in the same statement, precedence between them is from left to right.

Why AndAlso and OrElse in Addition to And and Or?

As previously discussed, the only difference between the And and AndAlso operators is that the AndAlso operator does not evaluate the second expression if the first expression is False. Similarly, the only difference between the Or and OrElse operators is that the OrElse operator does not evaluate the second expression if the first expression is True.

Not yet discussed is what difference does it really make whether you use And or AndAlso, or Or or OrElse?

The answer is there is no real difference if the second expression is simply a comparison, other than a slight savings in processor time for skipping the evaluation of the second expression. However, the second expression may be more complex, such as a function call that changes values. In that event, variables may have different values depending on whether the second expression was evaluated.

Conclusion

As programs become more sophisticated, they often branch in two or more directions based on whether a condition is True or False. For example, as discussed at the beginning of this chapter, a calculator program, before performing division, should check to see if the divisor is equal to zero, division by zero being illegal and if performed results in a run-time error. The program branches by performing the division if the divisor is not equal to zero, but warning the user if the divisor is equal to zero.

You use comparison operators to determine if the divisor is equal (or is not equal) to zero. There are comparison operators to test for equality or inequality, or whether one value is greater than or less than another.

A comparison operator can make only one comparison at a time, and sometimes you need to combine several comparisons. For example, to determine if someone is eligible to vote, you have to compare both their age to the minimum voting age and their country of citizenship to the United States. In this case, both comparisons must evaluate as True or the person is not allowed to vote. However, in other comparisons, only one of two conditions needs to be True. For example, you may be permitted to attend a movie without having to pay for a ticket if you are either a child or a senior citizen.

You use logical operators to combine several comparisons. The logical operators include And, when both comparisons must evaluate as True for an action to be taken, and Or, when only one of two comparisons must evaluate as True for an action to be taken. There are other logical operators as well.

The comparison and logical operators lay the groundwork for the following chapters, which use these operators to determine if a condition, or a combination of conditions, evaluates as True or False.

Quiz

1. What does the WriteLine method of the Debug class do?
2. What is the data type of the result of a comparison performed by a comparison operator?
3. How can you tell if the = operator is being used for assignment or comparison?
4. Can you use comparison operators with strings as well as with numeric data types?
5. What is the significance of Option Compare?
6. What does the Like operator do?
7. Which operators have precedence, comparison or arithmetic?
8. What is the purpose of a logical operator?
9. Which logical operator operates on only one operand rather than two?
10. Which operators have precedence, comparison or logical?



PART THREE

Controlling the Flow of the Program

This page intentionally left blank



CHAPTER

7

Making Choices—If and Select Case Control Structures

I showed you in Chapter 6 how to use comparison and logical operators to evaluate an expression as True or False. I will show you in this chapter how to use that information by employing control structures, specifically an If control structure or a Select Case control structure, so that different blocks of code execute depending on whether an expression evaluates as True or False.

The application user interacts with your code, including If and Select Case control structures, through the GUI of your application. You will learn in this chapter how to use two controls that often are utilized with If and Select Case control structures—the CheckBox and RadioButton controls.

The InputBox Function

Before discussing If and Select Case statements, let's discuss the input box, because it will be used in many of the code examples in this chapter.

Figure 7-1 shows an input box. It gets its name because the user may input text in the edit portion of the input box.

Modal vs. Modeless

The input box is *modal*. This means the application user cannot return to the main form without closing the input box, which is accomplished by choosing either the OK or the Cancel button.

Many forms in Windows applications are modal. Modal forms also are called *dialog forms*. One example is the Open dialog box shown in Figure 7-2, usually displayed by the File | Open menu command, and used to select and open a file in an application such as Microsoft Word. The Open dialog box is modal because you cannot return to the application before closing the dialog box, either by selecting a file and choosing Open, or by choosing Cancel.



Figure 7-1 The input box

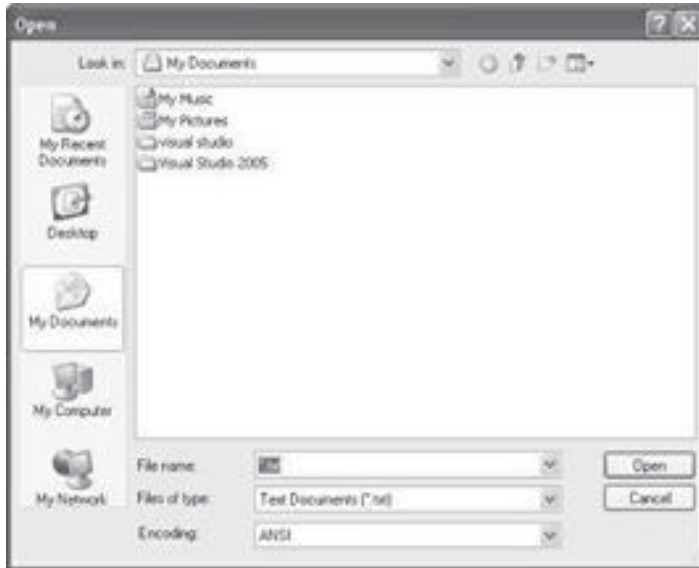


Figure 7-2 Modal Open dialog box

Not all forms are modal. For example, the Find form in Notepad, shown in Figure 7-3 and displayed with the Edit | Find menu command, is *modeless*, in that the application user can edit text in the main application without having to close the Find form.

Chapter 10 discusses modal dialog forms in more detail.



Figure 7-3 Modeless Find form

Displaying an Input Box

The input box is a form, but one built into Visual Basic, so you do not have to design it. Instead, you display it by calling the `InputBox` function. The syntax of the `InputBox` function is shown here:

```
InputBox( [Prompt], [Title], [Default Value])
```

Table 7-1 lists the parameters of the `InputBox` function.

The following code displays the input box shown in Figure 7-1 with a prompt for the application user to enter their name. Only the first, required parameter is used.

```
InputBox("Enter your name:")
```

Return Value

Choosing the OK button does more than close the input box and permit the application user to return to the main form. Choosing the OK button also returns a value of the `String` data type, the text the user entered in the input box.

The effect of returning a value is illustrated by the following code. If the user typed “George” in the input box and chose the OK button, the value “George” would be assigned to `strName`. Thus, the output of the `Debug.WriteLine` statement would be “George”.

```
Dim strName As String  
strName = InputBox("Enter your name: ")  
Debug.WriteLine(strName) 'outputs George
```

Name	Required?	Purpose
Prompt	Yes	The text informing the user what to enter, such as “Enter your name:” in Figure 7-1.
Title	No	The title of the input box. If this is omitted, the name of the project is used as the title, as in Figure 7-1.
Default Value	No	The value displayed in the input box when it is first displayed. If this is omitted, nothing (an empty string) is displayed, as in Figure 7-1.

Table 7-1 Parameters of the `InputBox` Function

Similarly, choosing the Cancel button does more than close the input box and permit the application user to return to the main form. Choosing the Cancel button also returns a value of the String data type, this time an empty string. An empty string has the value "" (two double quotes) because a string always is enclosed in double quotes, and there's nothing in between the two double quotes because the string is empty. Thus, in the preceding code, if the user chose the Cancel button, regardless of what they had entered in the input box, the value assigned to `strName` would be an empty string, and nothing would be output by the `Debug.WriteLine` statement.

The concept of returning a value works quite similarly to the previous examples of assignment statements. In an assignment statement, the value on the right side of the assignment operator is assigned to the variable or property on the left side of the assignment operator. Similarly, the value returned by a function on the right side of the assignment operator is assigned to the variable or property on the left side of the assignment operator.

In the preceding code example, the left side of the assignment statement was a String variable, but it also may be a property whose data type is String. The following code snippet assumes a Label control named `lblName`, whose `Text` property is assigned the text input by the user in the input box:

```
lblName.Text = InputBox("Enter your name: ")
```

If Control Structure

The If control structure comes in three varieties, depending on the number of alternative blocks of code:

- You use the If...Then statement if you want a block of code to execute if a condition is True but no block of code to execute if the condition is False. For example, if a purchaser is eligible for a senior citizen discount, you adjust the price, but if not, there is no price change to make.
- You use the If...Then...Else statement if you want one block of code to execute if a condition is True, and a second, different block of code to execute if the condition is False. This code structure often is used when there are two alternatives, such as Yes or No, or Male or Female.

- The If...ElseIf statement is similar to the If...Then...Else statement except that the If...ElseIf statement is used when there are more than two choices. For example, if your test score is 90 or better, your grade is an A; if your test score is between 80 and 89, your grade is a B; if your test score is between 70 and 79, your grade is a C, and so on.

If...Then Statement

You use an If...Then statement to execute code if, and only if, a condition is True. If the condition is False, then the code dependent on the If...Then statement does not execute. After the If...Then statement finishes executing, execution continues with the code, if any, following the statement.

The syntax of an If...Then statement is shown here:

```
If [condition] Then
    [Code]
End If
```

NOTE *There also is a one-line version with no End If at the end. However, that version does not work if there is more than one statement. Additionally, based on my teaching experience, I don't believe that one-line version is good practice anyway. Accordingly, I am using in this book the version with an End If at the end.*

Try the following code. It displays “You entered a positive number” to the Output window only if the input is a positive number (greater than or equal to zero). However, it displays “This line will always print” whether or not the input is a positive number, because after the If...Then statement finishes executing, execution continues with the code following the If...Then statement.

```
Private Sub Form1_Load(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles Me.Load
    Dim strScore As String
    Dim intScore As Integer
    strScore = InputBox("Enter a score")
    intScore = Integer.Parse(strScore)
    If intScore >= 0 Then
        Debug.WriteLine("You entered a positive number")
    End If
    Debug.WriteLine("This line will always print")
End Sub
```

The first `Debug.WriteLine` statement is indented to show that this statement will execute only if the `If` condition is `True`. This indenting is not required by the compiler. Rather, it is helpful to the programmer to see the flow of the code, and will be used in this and later chapters. Often the Visual Basic 2005 IDE will add the indentation for you.

NOTE *This code assumes that the user entered a number in the input box and clicked the OK button. Otherwise, an error would result. The “Input Validation” section later in this chapter will discuss how to guard against this error.*

The comparison may also use logical operators, as in the following code, which validates a test score as being between 0 and 100.

```
Private Sub Form1_Load(ByVal sender As Object, _  
    ByVal e As System.EventArgs) Handles Me.Load  
    Dim strScore As String  
    Dim intScore As Integer  
    strScore = InputBox("Enter a score")  
    intScore = Integer.Parse(strScore)  
    If intScore >= 0 And intScore <= 100 Then  
        Debug.WriteLine _  
            ("You entered a valid test score (0 - 100)")  
    End If  
    Debug.WriteLine("This line will always print")  
End Sub
```

This code displays “You entered a valid test score (0 - 100)” in the Output window only if the input is between 0 and 100. However, it displays “This line will always print” whether or not the input is between 0 and 100.

If...Then...Else Statement

You use the `If...Then...Else` statement if you want one block of code to execute if the condition is `True`, and a second, different block of code to execute if the condition is `False`. This differs from the `If...Then` statement in that some code in the `If...Then...Else` statement will execute; the only question is which. By contrast, with the `If...Else` statement, if the condition is `False`, then no code dependent on the `If...Then` statement executes.

After the `If...Then...Else` statement completes executing, execution continues with the code following the statement.

The syntax of an If...Then...Else statement is shown here:

```
If [condition] Then
    [Code]
Else
    [Code]
End If
```

No express condition follows the Else statement because the condition is implied as being the negation of the condition following the If statement. In other words, the code following the Else statement executes if the condition following the If statement is not True.

Try the following code. It displays in the Output window “You entered a valid test score (0 - 100)” if the input is between 0 and 100, but instead “You did not enter a valid test score” if the input is less than 0 or greater than 100.

```
Private Sub Form1_Load(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles Me.Load
    Dim strScore As String
    Dim intScore As Integer
    strScore = InputBox("Enter a score")
    intScore = Integer.Parse(strScore)
    If intScore >= 0 And intScore <= 100 Then
        Debug.WriteLine _
            ("You entered a valid test score (0 - 100)")
    Else
        Debug.WriteLine _
            ("You did not enter a valid test score")
    End If
    Debug.WriteLine("This line will always print")
End Sub
```

Although you can have an If without an Else, as with the If...Then statement, you cannot have an Else without an If. This is logical because Else means “none of the above,” and without an If there is no “above.”

If...ElseIf Statement

You use the If...ElseIf statement if you have more than two alternative blocks of code, the maximum possible with an If...Then...Else statement.

With an If...ElseIf statement, the first block of code whose condition is True executes, and all following blocks of code are skipped. The first block of code

follows the If clause, and each succeeding block of code coupled with a condition is an ElseIf clause. You can have as many ElseIf clauses as you want. Finally, you may optionally have an Else clause, which, as with an If...Then...Else statement, acts as “none of the above.” After the If...ElseIf statement finishes executing, execution continues with any code following the statement.

The syntax of an If...ElseIf statement is shown here:

```
If [condition] Then
    [Code]
ElseIf [condition] Then
    [Code]
Else
    [Code]
End If
```

Try the following code. It displays in the Output window “The test score is valid” if the input is between 0 and 100, “Test score cannot be less than zero” if the input is less than 0, or “Test score cannot be greater than 100” if the input is greater than 100.

```
Private Sub Form1_Load(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles Me.Load
    Dim strScore As String
    Dim intScore As Integer
    strScore = InputBox("Enter a score")
    intScore = Integer.Parse(strScore)
    If intScore >= 0 And intScore <= 100 Then
        Debug.WriteLine("The test score is valid")
    ElseIf intScore < 0 Then
        Debug.WriteLine _
            ("Test score cannot be less than zero")
    Else
        Debug.WriteLine _
            ("Test score cannot be greater than 100")
    End If
    Debug.WriteLine("This line will always print")
End Sub
```

Although you can have as many ElseIf clauses as you want, none can appear after an Else clause. The Else clause is optional; it serves the function of “none of the above.”

As is the case with the Else clause, although you can have an If without an ElseIf, you cannot have an ElseIf without an If.

Input Validation

The code used in the preceding section “If...ElseIf Statement” involves the entry of a student’s test score. No matter how badly a student performs on a test, that student will do no worse than 0. Similarly, no matter how well a student performs on a test, that student will do no better than 100.

However, it is not prudent to assume that the application user has entered a number between 0 and 100 in the input box and clicked the OK button. Human error is inevitable. An application user may not even read directions, much less follow them. Further, even a conscientious application user will make data-entry errors.

For example, if the application user entered in the input box a number less than 0 or larger than 100, that input necessarily is incorrect. If that incorrect input is stored as the student’s test score, then the student’s records will be wrong. Even worse, under the saying “garbage in, garbage out,” any calculation based on that test score also will be wrong.

Accordingly, your code should guard against the possibility that the application user’s input is incorrect. This is called validating the user’s input, or input validation.

The code used in the preceding section “If...ElseIf Statement” does perform input validation. The following portion of that code checks if the user’s input is between 0 and 100, and it warns the user if the input is incorrect:

```
If intScore >= 0 And intScore <= 100 Then
    Debug.WriteLine("The test score is valid")
ElseIf intScore < 0 Then
    Debug.WriteLine _
        ("Test score cannot be less than zero")
Else
    Debug.WriteLine _
        ("Test score cannot be greater than 100")
End If
```

Exceptions

The code used in the preceding section on the “If...ElseIf Statement” performs some input validation, but not enough. For example, that code does not guard against the possibility that the application user might enter a nonnumeric string and press the OK button. To demonstrate this, run the project with the Debug | Start Without Debugging menu command, or the CTRL-F5 menu command. When the input box displays, enter “Jeff” (without the quotes) and click the OK button of the input box. Your application will halt, and the message box shown in Figure 7-4 will display with the message “Input string was not in a correct format.”



Figure 7-4 An “Input string was not in a correct format” exception

Stop the project by clicking the Quit button. Run the project again with the Debug | Start Without Debugging menu command. When the input box displays, click the Cancel button of the input box. Your application will halt, and the message box shown in Figure 7-5 will display with the message “Value cannot be null. Parameter name: String.”

What Is an Unhandled Exception?

Figures 7-4 and 7-5 both refer to an “unhandled exception.” An *exception* is a problem that occurs while the program is executing that must be dealt with before the program can proceed. Examples of exceptions include the inability to open a file because it cannot be found, the application user did not put in the floppy drive the floppy disk that contains the file, the file is corrupt, the operating system does not have enough available memory remaining to open the file, and so on. The exception may be due to faulty code, application user error, or circumstances beyond the control of either the programmer or the application user, such as a crash of the operating system. Regardless of the cause, the program cannot proceed until the exception is resolved.

It is possible through code to “handle” an exception. For example, if the application user forgot to put in the floppy drive the floppy disk that contains the



Figure 7-5 A “Value cannot be null. Parameter name: String” exception

file, code warns the user and gives the user an opportunity either to put the floppy disk in the floppy drive or quit the application.

Exception handling is an advanced subject, so it's not covered here. For present purposes, exceptions generally do not crash programs, unhandled exceptions crash programs. That is why both Figures 7-4 and 7-5 refer to an "unhandled exception."

Determining Where the Exception Occurred

Although this explains what an unhandled exception is generally, what remains to be explained is what caused the unhandled exception in this code. You can determine the details of the exception by clicking the Details button. Figure 7-6 shows the result of clicking the Details button of the message box depicted in Figure 7-5.

One of the lines in the details is shown here:

```
at System.Int32.Parse(String s)
```

This matches the following line of our code:

```
intScore = Integer.Parse(strScore)
```

As mentioned in an earlier chapter, `Integer` is an alias in the .NET Framework for the `Int32` data type.

The reason for the error is that the `Parse` method of the `Integer` class requires for its parameter a string representation of an integer. Neither "Jeff" nor an empty

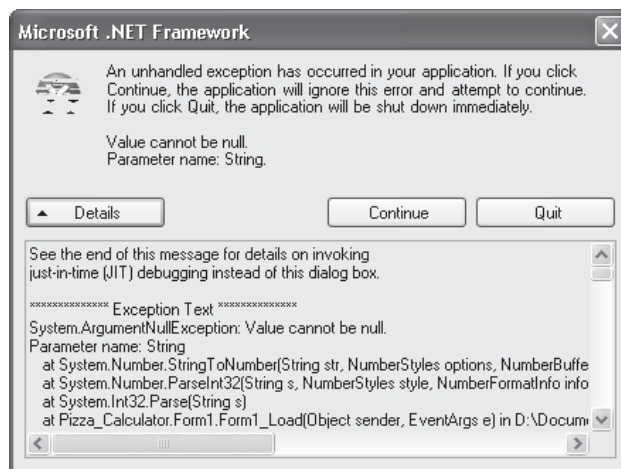


Figure 7-6 Exception details

string is a string representation of an integer. Therefore, the Parse method is unable to properly execute, and an exception occurs.

TryParse Method

The Integer class has a TryParse method in addition to a Parse method. Both methods convert the string representation of an integer into an integer. However, the TryParse method also returns a Boolean value (True or False) indicating whether the conversion was successful. If the conversion is not successful, such as because the argument is “Jeff” or an empty string, no exception occurs. Rather, the return value is False.

NOTE *Other numeric classes, such as Double, also have a TryParse method. In the case of the Double class, the method attempts to convert the string representation of a Double into a Double.*

The syntax of the TryParse method of the Integer class is shown here:

```
[Boolean] = Integer.TryParse([string], [integer])
```

The first parameter, the string, is the string representation of an integer. This argument usually is a variable, though it also could be a property of the String data type, such as the Text property of a Label control.

The second parameter, an integer, is where the integer equivalent of the string representation will be stored. This argument usually is a variable, though it also could be a property of the Integer data type.

The return value is Boolean and usually stored in a variable of that data type.

The following code snippet illustrates the TryParse method in action:

```
Dim strScore As String
Dim intScore As Integer
strScore = InputBox("Enter a score")
Dim blnInput As Boolean
blnInput = Integer.TryParse(strScore, intScore)
If blnInput = False Then
    ' Conversion unsuccessful.
    ' Don't use intScore in further code
Else
    ' Conversion successful.
    ' Use intScore in further code
End If
```

The following code implements this logic and modifies the code used in the preceding section “If...ElseIf Statement”:

```
Private Sub Form1_Load(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles Me.Load
    Dim strScore As String
    Dim intScore As Integer
    strScore = InputBox("Enter a score")
    Dim blnInput As Boolean
    blnInput = Integer.TryParse(strScore, intScore)
    If blnInput = False Then
        Debug.WriteLine _
            ("Input does not evaluate to an integer")
    ElseIf intScore >= 0 And intScore <= 100 Then
        Debug.WriteLine("The test score is valid")
    ElseIf intScore < 0 Then
        Debug.WriteLine _
            ("Test score cannot be less than zero")
    Else
        Debug.WriteLine _
            ("Test score cannot be greater than 100")
    End If
    Debug.WriteLine("This line will always print")
End Sub
```

Controls Used for If Control Structure

The application user interacts with your code, including the If control structure, through the graphical user interface (GUI) of your application. Two controls in particular are used in conjunction with the If control structure. The CheckBox control is used when a particular decision has only two choices, as in True or False, Yes or No, and so on. The RadioButton control is used when there are multiple, mutually exclusive choices, such as whether a student’s grade is an A, B, C, D, or F.

CheckBox Control

CheckBox controls are commonly used in Windows applications. For example, in the Print dialog box shown in Figure 7-7, there are check boxes for Print to File and Collate.

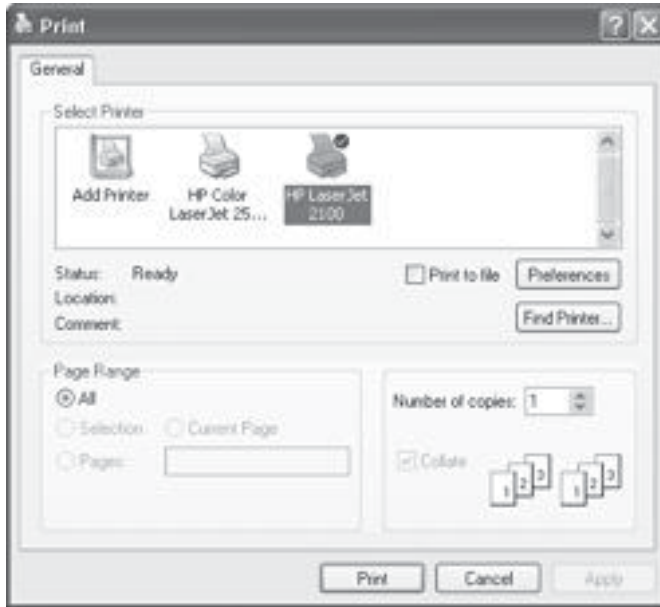


Figure 7-7 The Print dialog box

The reason that `CheckBox` controls are often used is that they are ideal for situations in which they are only two choices, such as Yes or No, Male or Female, and so on. The `CheckBox` control being checked is considered True or Yes or On, with unchecked being False or No or Off.

Each `CheckBox` control is independent of the others. They may all be checked, or all unchecked, or any combination of checked and unchecked.

The `CheckBox` control has two properties that you will use often: `Text` and `Checked`.

The `Text` property essentially is a label, built into the `CheckBox` control, that identifies to the application user the purpose of the check box. When you add the `CheckBox` control to the form, you have to draw it large enough (after first setting `AutoSize` to `False` in the Properties window) to show the text portion as well as the check box portion. The `Text` properties of the two `CheckBox` controls in Figure 7-7 are `Print to File` and `Collate`, respectively. The `Text` property usually is set at design time.

The `Checked` property is of a Boolean data type. If the check box is checked, the value of the `Checked` property is `True`. If the check box is not checked, the value of the `Checked` property is `False`.

Because the `Checked` property has only two possible values, `True` and `False`, often you use an `If...Else` statement based on the `Checked` property, as the following code snippet illustrates:

```
If chkPizza.Checked = True Then
    Debug.WriteLine "I want pizza!"
Else
    Debug.WriteLine "I don't want pizza."
End If
```

RadioButton Control

`RadioButton` controls also are commonly used in Windows applications. Taking again the example of the Print dialog box in Figure 7-7, there are radio buttons for printing all pages, just the current page, a range of pages, or the selected text.

The primary difference between `CheckBox` and `RadioButton` controls is that whereas each check box is independent of another, all radio buttons in a group are related in that only one of them can be chosen at any one time. Therefore, the `RadioButton` control is ideal for situations in which there are choices, but only one item can be chosen.

NOTE *If radio buttons are contained within a `GroupBox` or `Panel` control, then those radio buttons are a group independent of any other radio buttons on the form. This is useful when one set of radio buttons that, say, concerns age is logically independent of another set of radio buttons that concerns income level, for example.*

As with the `CheckBox` control, the two properties you will use often with the `RadioButton` control are `Text` and `Checked`. As with the `CheckBox` control, the `Checked` property for a `RadioButton` has only two possible values: `True` and `False`.

In the event you have more than two `RadioButton` controls, often you'll use an `If...ElseIf` statement based on the `Checked` property:

```
If radLarge.Checked = True Then
    Debug.WriteLine "I want a large pizza."
ElseIf radMedium.Checked = True Then
    Debug.WriteLine "I want a medium pizza."
Else
    Debug.WriteLine "I want a small pizza."
End If
```

Pizza Calculator

This project calculates the cost of the programmer's food of choice, pizza, using radio buttons and check boxes. The cost of the pizza is based initially on whether the pizza is a small (\$5.00), medium (\$7.50), or large (\$10.00). There is an additional cost of 50 cents for each topping.

Figure 7-8 shows the project in action. Because the application user has selected a large pizza (\$10.00) with pepperonis and anchovies (\$1.00 for two toppings), the total cost is \$11.00.

Creating the Project

Radio buttons are used to represent the three alternative pizza sizes: small, medium, and large. The radio buttons are named `radSmall`, `radMedium`, and `radLarge`, respectively. Similarly, their Text properties are, respectively, Small, Medium, and Large.

Check boxes are used to represent each topping choice: mushrooms, pepperoni, or my favorite, anchovies (because no one else wants anchovies, I get the whole pizza for myself). The check boxes are named `chkMushroom`, `chkPepperoni`, and `chkAnchovy`, respectively. Similarly, their Text properties are, respectively, Mushroom, Pepperoni, and Anchovy.

There are two Button controls. One is named `btnCalculate`, and its Text property is Calculate. The other is named `btnClear`, and its Text property is Clear.



Figure 7-8 The Pizza Calculator project

There also are two Label controls. The one that displays the total in Figure 7-8 is named `lblTotal`. Its `Text` property initially is blank. I also have set its `AutoSize` property to `False` and its `BackColor` property to `HighlightText` using the Properties window to give it its white background. The other label has a `Text` property of `Total`. It is not involved in the code, so you can retain its default name, such as `Label1`.

How the Project Works

The cost of the pizza is based initially on whether the pizza is a small (\$5.00), medium, (\$7.50), or large (\$10.00). There is an additional cost of 50 cents for each topping.

Clicking the Calculate button calculates and displays the cost in the Label control named `lblTotal`. Clicking the Clear button returns the application to its default settings (large size, all toppings unchecked, cost blank).

The Code

The code will be written in three places:

- Constants at module level to represent the cost of the pizza sizes and toppings
- Click event procedure of the Calculate button to calculate the cost of the pizza
- Click event procedure of the Clear button to restore the application to its default settings

Declaring the Constants

The following code is at module level, above and outside of the event procedures:

```
Const LARGE As Double = 10
Const MEDIUM As Double = 7.5
Const SMALL As Double = 5
Const TOPPING As Double = 0.5
```

These constants represent the costs of the different sizes of pizza and the extra cost of each topping. The actual values instead could have been used in the code. However, using constants makes the code easier to change if the costs of the different sizes or the toppings ever change, because only one change would need to be made (the value of the constant) rather than changing the value at all places it is used in

the code. Similarly, the constants `LARGE` and `SMALL` are declared as a `Double` instead of an `Integer` because someday the price may involve cents, such as the price of a large pizza changing from \$10.00 to \$10.50.

NOTE *These constants could have been declared in the Click event procedure of the Calculate button instead of in the General Declarations section because they only are referred to in the Click event procedure of the Calculate button.*

Calculating the Price

The following code is in the Click event procedure of the Calculate button:

```
Private Sub btnCalculate_Click _  
    (ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles btnCalculate.Click  
    Dim dblTotal As Double  
    If radLarge.Checked = True Then  
        dblTotal = LARGE  
    ElseIf radMedium.Checked = True Then  
        dblTotal = MEDIUM  
    Else  
        dblTotal = SMALL  
    End If  
    If chkMushroom.Checked = True Then  
        dblTotal += TOPPING  
    End If  
    If chkPepperoni.Checked = True Then  
        dblTotal += TOPPING  
    End If  
    If chkAnchovy.Checked = True Then  
        dblTotal += TOPPING  
    End If  
    lblTotal.Text = dblTotal.ToString("c")  
End Sub
```

Most of the work is done in the Click event procedure of the Calculate button. The variable `dblTotal` is used to store the total price. The data type of this variable is `Double` instead of `Integer` because the number may be a floating-point number (have cents as well as dollars).

An `If...ElseIf...Else` statement is used to assign to `dblTotal` the cost of the size of pizza selected, based on which radio button's value is `True`. An `If...ElseIf...Else` statement is appropriate because one, but only one, of the radio buttons can be selected.

By contrast, independent If statements are used to determine whether to add 50 cents for each topping, based on whether each check box's value is True. Independent If statements are appropriate because the value of each check box is independent from that of the others. The user may choose all toppings, no toppings, or any combination.

Finally, the value of `dblTotal` is displayed in the Total label. This involves two steps. First, the value is converted from a Double to a String data type using the `ToString` method because that value is being assigned to a property (Text) that is a String data type. Second, the argument "c" is passed to the `ToString` method so the total is formatted as currency, starting with the dollar sign (\$) and having two numbers, no more and no less, to the right of the decimal point.

NOTE The argument "c" to the `ToString` method is a format specifier. There are other format specifiers, such as "e" for exponential or scientific notation and "p" for percentage.

Restoring the Application to Its Initial Settings

Finally, the following code in the Click event procedure of the Clear button returns the application to its default settings (large size, all toppings unchecked, cost blank):

```
Private Sub btnClear_Click _  
    (ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles btnClear.Click  
    radLarge.Checked = True  
    radMedium.Checked = False  
    radSmall.Checked = False  
    chkMushroom.Checked = False  
    chkPepperoni.Checked = False  
    chkAnchovy.Checked = False  
    lblTotal.Text = ""  
End Sub
```

Select Case Control Structure

The Select Case control structure is quite similar to the If...ElseIf statement, but they are not the same. The primary difference is that, in the If...ElseIf statement, the If and ElseIf clauses each may evaluate completely different expressions, whereas

a Select Case control structure may evaluate only one expression, which then must be used for every comparison.

For example, the condition of an If clause could be whether `Night > Day` (the condition of the following ElseIf clause), whether `Citizenship = U.S.` (the condition of the next ElseIf clause), whether `NumberOfClasses >= 4`, and so on. Usually the conditions evaluated by the If and ElseIf clauses are related, but they can be completely independent of each other.

By contrast, the Select Case control structure evaluates one test expression, and that test expression is used for all the following comparisons.

Syntax

The syntax of a Select Case control structure is shown here:

```
Select Case [test expression]
Case [expression or expression list]
    [code]
' More Case statements optional
Case Else 'also optional
    [code]
End Select
```

The test expression may be a variable, constant, property, or expression.

The expression or expression list following the Case clause is compared to the expression or expression list, and may be one of the following:

- **An expression, such as Case 80** This means that the condition is whether the test expression equals the expression (in this example, whether the test expression equals 80).
- **An expression list, such as Case 80 To 90** This means that the condition is whether the test expression equals a value within the expression list (here, 80 through 90). If the values are not consecutive, then commas can delimit them. For example, Case 1 To 4, 7 To 9, 11 means that the condition is whether the test expression equals 1 through 4, or 7 through 9, or 11.

The Is Keyword

The Is keyword is combined with a comparison operator. For example, `Case Is > 8` means that the condition is whether the test expression is greater than 8.

These alternatives can be combined. For example, `Case 1 To 3, 5, Is > 8` means that the condition is whether the text expression is 1 through 3, 5, or greater than 8.

Select Case Control Structure in Action

Although the Select Case control structure differs from the If...ElseIf...Else statement in that it may evaluate only one expression that then must be used for every comparison, it otherwise behaves quite similarly to the If...ElseIf...Else statement.

If the condition following an If (or ElseIf) clause in an If...ElseIf...Else statement evaluates as True, the code following that clause executes, and none of the following ElseIf (or Else) clauses is evaluated. Similarly, if the expression or expression list following a Case clause matches the test expression, the code following the Case clause executes, and any remaining Case clauses are not evaluated.

If the condition following an If (or ElseIf) clause in an If...ElseIf...Else statement instead evaluates as False, the code following that clause does not execute, and each of the following ElseIf (or Else) clauses is evaluated in order. Similarly, if the expression or expression list following a Case clause does not match the test expression, the code following that clause does not execute, and each of the following Case clauses is evaluated in order.

If none of the conditions following the If and ElseIf clauses in an If...ElseIf...Else statement evaluates as True, the code following the Else clause executes if there is an Else clause. Similarly, if none of the conditions following the Case clauses in a Select...Case control structure matches the test expression, the code following the Case Else clause executes if there is a Case Else clause. The Case Else statement is analogous to the Else clause, covering the “none of the above” circumstance.

Once execution of the If...ElseIf...Else statement is completed, the program continues to the code following the End If statement. Similarly, once execution of the Select...Case control structure is completed, the code program continues to the code following the End Select statement.

Create a new Windows application and try running the following code. The user inputs a score. The Select Case control structure evaluates the value of that variable and then outputs either the grade based on that value or “Invalid score” if the score is not between 50 and 100.

```
Private Sub Form1_Load(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles Me.Load
    Dim intScore As Integer
    intScore = InputBox("Enter a score")
    Select intScore
        Case 90 to 100
            Debug.WriteLine "Your grade is an A"
        Case 80 to 89
            Debug.WriteLine "Your grade is a B"
        Case 70 to 79
            Debug.WriteLine "Your grade is a C"
```

```
Case 60 to 69
    Debug.WriteLine "Your grade is a D"
Case 50 to 59
    Debug.WriteLine "Your grade is an F"
Case Else
    Debug.WriteLine "Invalid score"
End Select
End Sub
```

Choosing Between If...ElseIf and Select Case

The If... ElseIf statement and the Select Case control structure are similar. However, in deciding whether to use If... ElseIf or Select Case, you may not have a choice.

Although any code you write using a Select Case control structure also can be written using an If statement, the reverse is not also true. If you need to evaluate several different expressions in a block of code, then you cannot use a Select Case control structure, which may evaluate only one expression that then must be used for every comparison.

If you do have a choice, the decision is one of personal preference, concerning which way is easier to write and easier to understand. Often your choice may be the Select Case control structure. Its structure often is the more readable of the two. Try writing the equivalent of Case 1 To 4, 7, 8 To 11, 14 in an If or ElseIf statement; you will have a very long list of comparisons joined by a number of And and Or operators.

Conclusion

In Chapter 6 you learned how to use comparison and logical operators to evaluate an expression as True or False. You learned in this chapter how to use that information by employing control structures, specifically an If or a Select Case control structure, so that different blocks of code execute depending on whether an expression evaluates as True or False.

The application user interacts with your code, including If and Select Case control structures, through the GUI of your application. You learned in this chapter how to use two controls that often are utilized with If and Select Case control structures: the CheckBox and RadioButton controls.

In the next chapter I will show you how to apply this information with loops, which enable you to repeat the execution of code statements.

Quiz

1. What does *modal* mean?
2. What is the converse of modal?
3. What is the return value of the `InputBox` function if the OK button is clicked?
4. What is the return value of the `InputBox` function if the Cancel button is clicked?
5. What are the three varieties of an If control structure?
6. What is an exception?
7. What does the `TryParse` method of the `Integer` class do?
8. Which two controls are commonly used with the If control structure?
9. What is the primary difference between the `If...ElseIf` statement and the `Select Case` control structure?
10. What part of a `Select Case` control structure performs the same purpose as an `Else` clause in an If control structure?

Repeating Yourself—Loops and Arrays

Parents customarily remind their children not to repeat themselves. Indeed, parents often illustrate another saying (“Do as I say, not as I do”) by continually repeating that reminder.

Sometimes you also want your code to repeat itself. For example, if the user enters invalid data, you may want to ask the user whether they want to retry or quit. If they retry and still enter invalid data, you again would ask the user whether they want to retry or quit. This process keeps repeating until the user either enters valid data or quits.

You use a loop to repeat the execution of code statements. A loop is a structure that repeats the execution of code until a condition becomes False. In the preceding example, the condition is that the data is invalid and the user should retry. The repeating code is the prompt asking the user whether they want to retry or quit and then permitting them to retry if they want to.

I will show you in this chapter the different types of loops available and how to implement them.

An array permits you to use a single variable to store many values. The values are stored at consecutive indexes, starting with zero and then incrementing by one for each additional element of the array. For example, to store sales for each day of the week, you can create one array with seven elements, rather than declaring seven separate variables. Using an array has several advantages. It is easier to keep track of one variable than seven. Additionally, you can use a loop to access each consecutive element in an array, whether to assign a value of that element or to display that value.

I will show you in this chapter how to create and use arrays.

Loops

This section will introduce three types of loops: For...Next, While...End While, and Do.

For...Next Statement

A For...Next statement generally is used to repeat the execution of a statement a fixed number of times.

Syntax

The syntax of a For...Next statement is shown here:

```
For counter As DataType = start To end [ Step increment ]  
    [ statements ]  
Next [ counter ]
```

The variable following the For keyword normally is referred to as a “counter” because its value determines the count of the number of loops. The data type of the counter variable would be an Integer or other whole number data type.

If the counter variable was declared before the loop, then its data type would not be declared in the For...Next statement. However, because counter variables normally are not used outside of loops, they often will be declared in the For...Next loop as in the preceding syntax and the following examples.

The next code example illustrates this syntax:

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



```
For x As Integer = 1 To 5
    Debug.WriteLine(x)
Next x
Debug.WriteLine("This line will always print")
End Sub
```

This code will display the following in the Output window:

```
1
2
3
4
5
This line will always print
```

Where Is the Step Statement?

When you compare the code to the syntax of the For...Next statement, you legitimately may wonder, Where is the Step statement?

The answer is that when the Step statement is left out, the default is Step 1. Therefore, the following two lines of code are the same:

```
For x = 1 To 5
For x = 1 To 5 Step 1
```

NOTE *In following sections we will look at Step statements having different values than the default 1.*

Now that we have accounted for the Step statement, let's analyze the other parts of the For...Next statement.

Where's the Comparison Operator?

As mentioned in the introduction, a loop is a structure that repeats the execution of code until a condition becomes False. A condition usually involves the comparison of a variable with a value. In a For...Next statement, the comparison is done with a comparison operator, either \geq or \leq .

You won't see a comparison operator in the For...Next statement. Instead, if the value following the Step statement is positive, the comparison operator is \geq . Conversely, if the value following the Step statement is negative, the comparison operator is \leq .

Because in this code there is no Step statement, the default, Step 1, applies. Because 1 is of course positive, the comparison operator is \geq .

NOTE In following sections we will look at Step statements in which the value is negative and therefore the comparison operator is $>=$.

Which Values Are Being Compared?

An operator such as $>=$ compares two values. The value on the left side of the comparison usually is contained in a variable that immediately follows the For keyword. That variable, here the Integer variable x , often is referred to as a “counter.”

The value on the right side of the comparison operator is the end value. In the following line from the code, the end value is 5:

```
For x = 1 To 5
```

Because the Step value, 1, is positive, the comparison is $x \leq 5$.

Finally, the initial or starting value of the counter variable x is 1, the value that is assigned to it ($x = 1$) immediately following the For keyword. Of course, that value must change, or the comparison $x \leq 5$ will always be the same, resulting in the loop executing infinitely. The upcoming section “Execution of the Loop” will discuss how the value of x will change.

Dependent Code

A loop is a structure that repeats the execution of code until a condition becomes False. The code whose repeated execution is dependent on the condition being True is between the line beginning with For and the line beginning with Next. That code could be only one statement, or multiple statements. In this example, the code whose execution is depending on $x \leq 5$ being True is only one statement: `Debug.WriteLine(x)`.

By contrast, the line of code `Debug.WriteLine(“This line will always print”)` is after the line beginning with Next. Therefore, that line of code is not part of the For...Next statement and will execute regardless of whether $x \leq 5$ is True.

Execution of the Loop

As discussed in the earlier section “Which Values Are Being Compared?”, the condition is $x \leq 5$, and the starting value of x is 1. Accordingly, the condition $x \leq 5$ initially is True, because $1 \leq 5$.

Because the condition is True, the dependent code, `Debug.WriteLine(x)`, will execute, and the value of `x`, currently 1, is printed to the Output window.

Execution next reaches the line beginning with `Next`. This line causes the value of `x` to change by the value following the `Step` keyword. As discussed in “Where Is the Step Statement?”, because there is no `Step` statement, `Step 1` is implied, which means that the value of `x` will increase by 1 to 2.

Execution now returns to the line beginning with the `For` keyword. The condition `x <= 5` still is True because `2 <= 5`. Accordingly, the value of `x`, now 2, is printed to the Output window, and then the value of `x` is increased by 1 to become 3.

The condition `x <= 5` still is True because `3 <= 5`. Accordingly, the value of `x`, now 3, is printed to the Output window, and then the value of `x` is increased by 1 to become 4.

The condition `x <= 5` still is True because `4 <= 5`. Accordingly, the value of `x`, now 4, is printed to the Output window, and then the value of `x` is increased by 1 to become 5.

The condition `x <= 5` still is True because `5 <= 5`. Accordingly, the value of `x`, now 5, is printed to the Output window, and then the value of `x` is increased by 1 to become 6.

When the value of `x` becomes 6, the condition `x <= 5` is False. Therefore, the code `Debug.WriteLine(x)` does not execute, so the value of `x`, 6, is not printed to the Output window. Additionally, the implied `Step 1` statement does not execute, so the value of `x` is not increased by 1, but rather remains 6.

When the `For...Next` statement stops executing, execution continues with the code that follows the `For...Next` statement, which is `Debug.WriteLine` “This line will always print”.

Table 8-1 summarizes the execution of the loop.

Value of x	<code>x <= 5</code>	Value of x Prints?	New Value of x
1	True	Yes	2
2	True	Yes	3
3	True	Yes	4
4	True	Yes	5
5	True	Yes	6
6	False	No	6

Table 8-1 Summary of Execution of `For...Next` Loop

Iteration

The dependent code inside the For...Next statement executed five times in the previous example. Each time the condition is True and the dependent code executes is referred to as an “iteration.” Accordingly, there were five iterations of the loop in the previous example.

Step Statement

If you want the value of the counter variable to change by a value other than 1, then you need to include a Step statement.

The following code modifies the previous by including a Step 2 statement. Accordingly, the values printed to the Output window will not be 1, 2, 3, 4, and 5 as in the previous example, but instead 1, 3, 5, followed of course by the line “This line will always print”:

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

    For x As Integer = 1 To 5 Step 2
        Debug.WriteLine(x)
    Next x
    Debug.WriteLine("This line will always print")
End Sub
```

Because the value of increment following the Step statement is 2, the value of x will increase by 2 every time the For...Next statement executes, until x is greater than 5, when the For...Next statement will stop executing. Thus, the first value of x that prints is its start value, 1, then 3, then 5. When the value of x becomes 7, it has passed the end value, 5, so the For...Next statement stops executing, and execution continues with the code that follows the For...Next statement.

So far the value following the Step statement has been a positive number, but it may be a negative number instead. Try the following code:

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

    For x As Integer = 1 To 5 Step -1
        Debug.WriteLine(x)
    Next x
    Debug.WriteLine("This line will always print")
End Sub
```

This code will compile, but it does not print any numbers to the Output window, just the line “This line will always print” that follows the conclusion of the For...Next statement. The reason why is that when the value following the Step statement is positive, the For...Next statement will execute so long as the counter variable \leq end. However, when the value following the Step statement is negative, the For...Next statement will execute so long as the counter variable \geq end, or here, $x \geq 5$. Because when the For...Next statement starts, the value of x is 1, and $1 \geq 5$ is False, the code inside the For...Next statement never executes.

If we want this code to print numbers to the Output window, we should switch the values of start and end so the code is as follows:

```
Private Sub Form1_Load(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles Me.Load  
  
    For x As Integer = 5 To 1 Step -1  
        Debug.WriteLine(x)  
    Next x  
    Debug.WriteLine("This line will always print")  
End Sub
```

This code will print to the Output window 5 through 1, in descending order, followed by “This line will always print.”

Exit For Statement

The Exit For statement transfers control immediately to the statement following the Next statement. Stated another way, the Exit For statement prematurely ends the execution of the For...Next statement before the condition becomes False.

The Exit For statement often is used in combination with the evaluation of a condition by an If...Then...Else statement. For example, the following code will output only 1 through 3, not 1 through 5, because the loop ends prematurely when x equals 4:

```
Private Sub Form1_Load(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles Me.Load  
  
    For x As Integer = 1 To 5  
        If x > 3 Then  
            Exit For  
        End If  
        Debug.WriteLine(x)  
    Next x  
    Debug.WriteLine("This line will always print")  
End Sub
```

Nesting

As the preceding example in the section on the Exit For statement shows, you can nest an If...End If statement in a For...Next statement. Try the following code, which will print each number between 1 and 100 that is evenly divisible by 3:

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

    Debug.WriteLine _
        ("Every multiple of 3 between 1 and 100:")
    For x As Integer = 1 To 100
        If x Mod 3 = 0 Then
            Debug.WriteLine (x)
        End If
    Next x
    Debug.WriteLine ("This line will always print")
End Sub
```

You also can nest For...Next loops by placing one loop within another. Each loop must have a different counter variable. Try the following code:

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

    For I As Integer = 1 To 3
        For X As Integer = 1 To 3
            Debug.WriteLine ("I = " & I)
            Debug.WriteLine ("X = " & X)
        Next X
    Next I
    Debug.WriteLine ("This line will always print")
End Sub
```

The output is shown here:

```
I = 1
X = 1
I = 1
X = 2
I = 1
X = 3
I = 2
X = 1
I = 2
X = 2
```

```
I = 2
X = 3
I = 3
X = 1
I = 3
X = 2
I = 3
X = 3
```

This line will always print

Nesting For...Next loops often is used to print values in the rows and columns of a table. The outside For...Next loop represents the rows, and the inside For...Next loop represents the columns.

While...End While Statement

The While...End While statement repeats the execution of a statement as long as a given condition is True. The syntax is shown here:

```
While condition
    [ statements ]
End While
```

The following code example shows a While...End While statement in action. It prints 1 through 5 to the Output window:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Me.Load
    Dim x As Integer = 1
    While x <= 5
        Debug.WriteLine(x)
        x += 1
    End While
    Debug.WriteLine("This line will always print")
End Sub
```

The condition is $x \leq 5$. If that condition is True, all the statements within the While...End While statement are executed. When the End While statement is reached, execution returns to the While statement and the condition is again checked. If the condition is still True, the process is repeated. If it is False, execution resumes with the statement following the End While statement.

The condition is $x \leq 5$. The starting value of x is 1, so at the beginning, the condition $x \leq 5$ is True. Therefore, both statements within the While...End While statement are executed; the value of x , at this time 1, is outputted, and then is incremented so the new value of x is 2.

When the End While statement is reached, execution returns to the While statement and the condition is again checked. The value of *x* is now 2, so the condition still is True because $2 \leq 5$. Accordingly, once again the value of *x* is outputted and then increased by 1 to become 3. The End While statement is reached, so execution returns to the While statement and the condition is again checked. The value of *x* is now 3, so the condition still is True because $3 \leq 5$. Accordingly, once again the value of *x* is outputted and then increased by 1 to become 4.

The End While statement again is reached, so execution returns to the While statement and the condition is checked once again. The value of *x* is now 4, so the condition still is True because $4 \leq 5$. Accordingly, once again the value of *x* is outputted and then increased by 1 to become 5. The End While statement is reached, so execution returns to the While statement and the condition is again checked. The value of *x* is now 5, so the condition still is True because $5 \leq 5$. Accordingly, once again the value of *x* is outputted and then increased by 1 to become 6.

Now when execution returns to the While statement and the condition is checked, the condition no longer is True because $6 \leq 5$ is False. Accordingly, the statements within the While...End While statement are not executed. Instead, execution continues with the code that follows the While...End While statement (“This line will always print”).

Differences Between For...Next and While...End While Loops

One significant difference between a While...End While statement and a For...Next statement is that in a While...End While loop, you have to affirmatively change the value of the variable (here *x*) used in the comparison that determines whether the loop continues to execute. By contrast, in a For...Next loop, the Step statement, whether express or implied, takes care of that detail for you.

Therefore, with a While...End While statement, you would have an infinite loop if you did not change the value of *x* in the preceding While...End While statement. In other words, instead of writing

```
While x <= 10
    Debug.WriteLine (x)
    x += 1
End While
```

you left out the `x += 1` line, as shown here:

```
While x <= 10
    Debug.WriteLine (x)
End While
```

The loop would not stop until you stopped the application.

Another difference between a While...End While statement and a For...Next statement is that a For...Next statement generally is intended to run a fixed number of times, whereas a While...End While statement may run an indefinite number of times. For example, if you want a menu to display until the user chooses the option to quit, the While...End While statement would be a better choice than the For...Next statement because the programmer could not predict how many times the user would choose to continue before selecting to quit.

Similarities Between For...Next and While...End While Loops

Although there are significant differences between a While...End While statement and a For...Next statement, there also are similarities. A While...End While statement, like a For...Next statement, may never execute the statements inside it if initially the condition is False. In the following example, no numbers will be written to the Output window because the condition initially is False:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Me.Load
    Dim x As Integer = 1
    While x >= 5
        Debug.WriteLine(x)
        x += 1
    End While
    Debug.WriteLine("This line will always print")
End Sub
```

You also can nest a While...End While statement inside another While...End While statement, just as you can nest a For...Next statement inside another For...Next statement.

Additionally, the Exit While statement serves the same function in a While... End While statement as the Exit For statement does in the For...Next statement, ending the While...End While loop prematurely. The following code will only output 1 through 3, not 1 through 5, to the Output window:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Me.Load
    Dim x As Integer = 1
    While x <= 3
        If x = 4 Then
            Exit While
        End If
        Debug.WriteLine (x)
        x += 1
    End While
End Sub
```

```
End While
Debug.WriteLine ("This line will always print")
End Sub
```

Do Statement

The Do statement, like the For...Next and While...End While statements, is used to repeat the execution of a statement. However, the Do statement comes in two varieties—one testing a condition at the top of the statement, the other at the bottom.

Testing a Condition at the Top of a Do Statement

The syntax of a Do loop that tests the condition at the top of the statement is shown here:

```
Do { While | Until } condition
    [ statements ]
Loop
```

You can use either While or Until, but you must use one (unless you have code inside the loop that will end it), and you cannot use both in the same statement.

For example, the following code uses the While keyword to output to the Output window 1 through 5, followed by “This line will always print”:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Me.Load
    Dim x As Integer
    x = 1
    Do While x <= 5
        Debug.WriteLine(x)
        x += 1
    Loop
    Debug.WriteLine("This line will always print")
End Sub
```

The same result could be achieved by using the Until keyword:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Me.Load
    Dim x As Integer
    x = 1
    Do Until x > 5
        Debug.WriteLine(x)
        x += 1
    Loop
```

```
Loop
  Debug.WriteLine("This line will always print")
End Sub
```

Whether you use `While` or `Until` is a matter of choice, depending on which is more intuitive to you.

Whether you use `While` or `Until`, it is important not to forget the statement `x += 1` in the preceding code. If the value of `x` does not change, the loop will never stop (you can still halt the program using the `CTRL-BREAK` keyboard combination). This is not such an important issue with a `For...Next` statement, where if a `Step` statement is omitted, a `Step 1` statement is the default, increasing the value of the counter by 1 each time the loop executes. As with the `While...End While` statement, there is no similar default with a `Do` statement.

The `Exit Do` statement performs the same purpose in a `Do` statement as the `Exit For` statement does in a `For...Next` statement and the `Exit While` statement does in a `While...End While` statement, prematurely ending execution of the loop.

Testing a Condition at the Bottom of a Do Statement

The variation of the `Do` statement that tests the condition at the top of the loop acts essentially the same as a `While...End While` statement. However, the other variation of the `Do` statement, which tests the condition at the end of the loop, is unique. Its syntax is shown here:

```
Do
  [ statements ]
Loop { While | Until } condition
```

With this syntax, the statements inside the loop will always execute at least once, because the first test is at the bottom of the loop after the statements. A menu is one example of when you may want the statements inside the loop to execute at least once. Try the following code, which requires the user either to input a numeric value or quit by clicking the `Cancel` button (which returns an empty string):

```
Private Sub Form1_Load(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles Me.Load
  Dim strInput As String
  Dim intInput As Integer
  Dim blnInput As Boolean
  Dim strMsg As String
  strMsg = "Enter a number. Quit by clicking Cancel"
  Do
    strInput = InputBox(strMsg)
    blnInput = Integer.TryParse(strInput, intInput)
```

```
If strInput = "" Then
    Debug.WriteLine _
        ("Nothing entered or Cancel selected")
ElseIf btnInput = False Then
    Debug.WriteLine("You need to enter a number")
Else
    Debug.WriteLine _
        ("You entered the number " & strInput)
End If
Loop Until btnInput = True Or strInput = ""
Debug.WriteLine("This line will always print")
End Sub
```

The test in this example should be at the bottom rather than at the top of the loop because the user has to enter a value before there is anything to test.

For Each...Next Loop

The For Each...Next loop is similar to the For...Next loop, but it executes the statement block for each element in a collection, instead of a specified number of times. A collection is a group of usually like objects. The syntax is shown here:

```
For Each [variable] [As Data Type] In [Collection]
    'code
Next [variable]
```

For example, a form has a Controls collection, which is a collection of all the controls on a form. The following code displays in the Output window the name of each control in the form:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Me.Load
    For Each ctl As Control In Me.Controls
        Debug.WriteLine(ctl.Name)
    Next ctl
End Sub
```

Arrays

In previous chapters, I showed you how to declare variables of different data types such as Integer or Double. Those variables are scalar variables. They can store only one value at a time.

An array permits you to use a single variable to store many values. The values are stored at consecutive indexes. The index is a positive integer, starting with zero and then incrementing by one for each additional element of the array.

Declaring Arrays

Array variables are declared the same way as other variables, with one difference. The array name is followed by a pair of parentheses, and within the parentheses you indicate the highest index of the array. For example, you would declare an array of seven Integers, each element representing sales for a day of the week, as follows:

```
Dim arrSalesPerDay(6) As Integer
```

The number 6 within the parentheses may appear confusing because the array has seven elements. However, the number within the parentheses is not the number of elements in the array. Rather, it is the upper bound, or highest index of the array. Because the number of the lowest index always is 0, the number of elements always is one more than the highest index.

Default Value

When you first declare an array, each element of the array has a default value. The specific default value depends on the data type of the array. If, as here, the data type is Integer, each element of the array has a default value of 0. Try the following code, which loops through the elements of the array and outputs seven zeros to the Output window:

```
Private Sub Form1_Load(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles Me.Load  
    Dim arrSalesPerDay(6) As Integer  
  
    For x As Integer = 0 To 6  
        Debug.WriteLine(arrSalesPerDay(x))  
    Next x  
End Sub
```

You can assign a value to an element of an array by using the index of the element. For example, the following code fragment assigns 73 to the second element of the array:

```
arrSalesPerDay(1) = 73
```

However, you can use a loop to efficiently assign values to each element of the entire array. Try the following code, which uses two loops. The first loop has the

user enter values for each day's sales. The second loop outputs the values the user entered:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Me.Load
    Dim arrSalesPerDay(6) As Integer
    Dim strTemp As String

    For x As Integer = 0 To 6
        strTemp = InputBox("Enter a day's sales")
        arrSalesPerDay(x) = Integer.Parse(strTemp)
    Next x
    For x = 0 To 6
        Debug.WriteLine(arrSalesPerDay(x))
    Next x
End Sub
```

NOTE *In this example, the lower and upper bounds of the array, 0 and 6, respectively, were known. You can obtain these values programmatically with the `GetLowerBound` and `GetUpperBound` methods, and you can obtain the number of elements in the array with the `Length` property.*

You also can use arrays with loops to obtain a running total. Try the following code, which outputs the total of the seven daily sales amounts entered by the user:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Me.Load
    Dim arrSalesPerDay(6) As Integer
    Dim strTemp As String

    Dim total As Integer = 0
    For x As Integer = 0 To 6
        strTemp = InputBox("Enter a day's sales")
        arrSalesPerDay(x) = Integer.Parse(strTemp)
    Next x
    For x = 0 To 6
        total += arrSalesPerDay(x)
    Next x
    Debug.WriteLine("Total Sales: " & total)
End Sub
```

As you can see, loops are very useful with arrays.

NOTE *The arrays in this chapter have one dimension. You can have arrays with two or more dimensions, two often representing rows and columns in a table or spreadsheet, three a cubic space, and so forth.*

Conclusion

Loops are used to repeat the execution of code statements. The For...Next statement is used to repeat code execution a fixed number of times. The While...End While statement is more flexible than the For...Next statement because the number of times a While...End While statement repeats does not have to be determined when you write the code, but may depend on user input. The Do statement has the additional flexibility that it may test the condition at the bottom rather than only at the top of the loop.

An array permits you to use a single variable to store many values. The values are stored at consecutive indexes, which start with zero and end at an index that is one less than the number of elements in the array.

In the next chapter, you will learn how to use subroutines and functions to organize your code more efficiently.

Quiz

1. What is a loop?
2. What is a difference between a While...End While statement and a For...Next statement?
3. What is a difference between the Do statement and the For...Next and While...End While statements?
4. What is a difference between the For Each...Next loop and the For...Next loop?
5. What are examples of nesting?
6. What does an array variable permit you to do that a scalar variable does not?
7. What is the difference between declaring an array variable and a scalar variable?

8. What is the lowest index of an array?
9. What is the relationship between the number of elements in an array and the highest index in that array?
10. If you declare an array without assigning a value to its elements, do its elements have a default value?

Organizing Your Code with Procedures

A procedure is a block of one or more code statements that execute when called upon to do so, whether by an event, code, or the .NET Runtime. Most Visual Basic code consists of procedures.

Visual Basic has many built-in procedures. One example is the `InputBox` function, which displays an input box and returns the text the user enters in the input box. Additionally, the definitions (name, parameters, return value, or lack thereof) of event procedures, such as the `MouseMove` event procedure of a form and the `Click` event procedure of a button, are built in, though of course you have to write the code that goes inside of an event procedure.

Additionally, you can write your own procedures. This chapter will explain two different types of procedures—subroutines and functions—and how to write them. This chapter also will explain how to call a procedure, necessary so the code within it will execute, how to pass information to a procedure, and how, in the case of a function, to return information to the code that called it.

Types of Procedures

This chapter will cover two important types of procedures:

- **Subroutine** Contains code that performs actions, but does not return a value to the event or code that calls it. Event procedures are subroutines.
- **Function** The same as a subroutine except that a function returns a value to the code that calls it. The `InputBox` function is an example of a function.

Because almost all Visual Basic code must be written within procedures, we necessarily have been working with procedures, specifically subroutines and functions, since the early chapters.

We started working with event procedures, procedures that contain a block of code that executes when an event happens to an object. Event procedures are subroutines. Although the code within them executes, they do not return a value as a function does. Later we discussed a procedure that does return a value, the `InputBox` function, which returns the value of text input by the application user.

Built-In vs. Programmer-Defined Procedures

Event procedures and the `InputBox` function have in common that they are built into Visual Basic or the underlying .NET Framework.

The code inside the event procedure executes, seemingly automatically, when the specified event happens to the specified object. You need not write any code to cause the event procedure to execute when the specified event happens; code provided automatically by the IDE does this for you. For example, the code inside the `Click` event procedure of the form executes when you click the form.

Although you do not need to write any code to cause an event procedure to execute, you do need to write code for the `InputBox` function to execute. However, you do not need to write any code for the `InputBox` function to do what it does—that is, display an input box and return the value typed by the application user in the input box.

Built-in procedures simplify your programming tasks. For example, you could write a procedure that duplicates what the `InputBox` function does. However, Visual Basic saves you the trouble by providing the `InputBox` function for you. Visual Basic has many other built-in procedures that simplify your programming tasks.

Although Visual Basic has numerous useful built-in procedures, not even the creators of Visual Basic could anticipate every conceivable programming task and supply a built-in procedure to perform that task. Indeed, even if they could, the Visual Basic language might become too large and unwieldy. Therefore, many times you will want to create your own procedures. Visual Basic enables you to do so, and this chapter will show you how.

Methods Contrasted

We also have been using methods. For example, we have used the `WriteLine` method of the `Debug` class to output, to the Output window, the value of its argument.

Methods are procedures, but not all procedures are methods. The primary difference between a method and a procedure is that a method may only be called from a specific class or object, whereas a procedure may be called independently from a class or object.

For example, the `WriteLine` method belongs to the `Debug` class. You could not call the `WriteLine` method from a `Form` object, or independently of any object. That is why the `Debug` class and the dot (`.`) operator precede the `WriteLine` method. By contrast, an event procedure, although it relates to a particular object, is not called from that object.

Subroutines

There are two steps to using a subroutine. The first step is to create it, by declaring the subroutine, much as you create a variable by declaring it. The second step is to call the subroutine so the code within it executes. Additionally, you can pass information to the subroutine by using arguments.

Declaring a Subroutine

Event procedures are built into Visual Basic and the underlying .NET Framework. Therefore, you do not need to tell Visual Basic what these procedures are.

However, if you write your own procedures, you do need to tell Visual Basic what they are. Otherwise, the compiler will not be able to recognize, much less execute, the procedure. The result at best will be that nothing happens, or worse, a compiler error.

As you tell Visual Basic what a variable is by declaring it, you similarly tell Visual Basic what a procedure is and does by declaring the procedure.

The syntax for declaring a subroutine is shown here:

```
[Accessibility] Sub name[(Parameter list)]
    Statements
    [Return | Exit Sub]
End Sub
```

Table 9-1 lists and describes the elements of declaring a subroutine.

The following programmer-defined subroutine, named PrintInput, illustrates this syntax. This subroutine outputs to the Output window what the user types in an input box, unless the user either did not type anything or chose the Cancel button of the input box:

```
Private Sub PrintInput ()
    Dim strInput As String
    strInput = InputBox("Enter something")
    If strInput = "" Then
        Return ' or Exit Sub
    End If
    Debug.WriteLine(strInput)
End Sub
```

Element	Required or Optional	Description
Accessibility	Optional	Determines where the subroutine may be called from. See Table 9-2.
Sub	Required	Keyword indicating the procedure is a subroutine.
Name	Required	Name of the subroutine.
Parameter list	Optional	Information passed to the subroutine.
Statements	Required	Code that executes each time the subroutine is called.
Return	Optional	Ends execution of the subroutine before the End Sub statement. Alternative to Exit Sub.
Exit Sub	Optional	Ends execution of the subroutine before the End Sub statement. Alternative to Return.
End Sub	Required	Ends execution of the subroutine.

Table 9-1 Elements of a Subroutine

You can type this code either before or after any event procedure, as long as it is before the End Class statement.

The accessibility specifier is Private. The keyword Sub indicates the procedure is a subroutine. The name of the subroutine is PrintInput. This subroutine has no arguments, so the parentheses are empty.

Turning to the body of the PrintInput subroutine, if strInput is an empty string, because either the user did not input anything or chose the Cancel button of the input box, then the Return statement, or alternatively an Exit Sub statement, ends the execution of the subroutine, so the Debug.WriteLine(strInput) statement does not execute. Otherwise, that statement does execute, and the subroutine then continues, and ends, with the End Sub statement.

Accessibility

Table 9-2 lists the accessibility specifiers for a procedure. The accessibility specifier is optional. If omitted, the accessibility specifier is Public.

These accessibility specifiers have essentially the same function with procedures as they do with module-level variables. Therefore, if you declare the procedure within a form and will only access the procedure from that form, you should declare the procedure as Private. However, if, for example, your project has multiple forms, and you plan to call the procedure declared in one form from another form, you might declare the procedure as Friend.

TIP As with module-level variables, you should declare procedures as Private unless you really need to be able to access them outside the class.

Naming the Procedure

You have relative freedom in naming a procedure as you do in naming variables. There are only a few limitations, such as no embedded spaces within the procedure name. For example, Print Input is not a valid procedure name.

Declared Accessibility	Meaning
Public	The procedure may be called from the current project or essentially any other project.
Protected	Access is limited to the class in which the variable was declared or the classes inherited from that class.
Friend	Access is limited to the classes in the current assembly (or solution).
Protected Friend	Combines access for Protected and Friend.
Private	Access is limited to the class in which the variable was declared.

Table 9-2 Accessibility Specifiers

Although Visual Basic imposes few limitations on how you name a procedure, as with naming variables, you should name procedures so that what they do is reasonably clear to you and other programmers who may have to review your code. Procedure names such as Sub1, Sub2, Sub3, and so on are not very helpful. You, and even more so your fellow programmers, will have trouble remembering which of them does what. By contrast, descriptive procedure names such as PrintName, PrintAddress, PrintCity, and so on, are quite helpful in describing what each procedure does.

I agree with Microsoft's recommendation that you use the *NounVerb* or *VerbNoun* style to create a name that clearly identifies what the procedure does. For example, the procedure name PrintName is a concatenation of the verb Print, which indicates the action the procedure takes, and the noun Name, which indicates the information printed. You might have more than one noun, such as PrintCustomerName. In any event, the first letter of each noun and verb is capitalized.

Parameter List

The PrintInput subroutine has no parameters. The subject of parameters—information given to a procedure that helps it perform its task—will be covered in this chapter as soon as we have covered the subject of calling the subroutine. This order will make the concept of parameters easier to understand. Accordingly, for now, the procedures we will use have no parameters.

Return and Exit Sub Statements

The Return statement ends execution of a subroutine before the End Sub statement. In the example PrintInput subroutine, if the Return statement executes (because the user did not type anything), then it ends the execution of the subroutine, so the Debug.WriteLine(strInput) statement does not execute.

Usually the Return statement is coupled with an If statement so that whether the Return statement executes depends on a condition that evaluates to True or False.

The Exit Sub statement accomplishes the same result as the Return statement. The two statements are interchangeable in subroutines.

Calling the Subroutine

Firefighters put out fires. However, they generally do not drive around looking for fires. Instead, they go out to a fire when called upon to do so.

In the same way, a subroutine does not just execute by itself. The statements within a subroutine do not execute until and unless the subroutine is called.

A subroutine usually is called by code. The following code example calls the `PrintInput` subroutine from the `Click` event procedure of the form:

```
Private Sub Form1_Click(ByVal sender As Object, _  
    ByVal e As System.EventArgs) Handles Me.Click  
    PrintInput()  
    Debug.WriteLine("When does this line display?")  
End Sub
```

Calling a subroutine starts with the subroutine's name (here, `PrintInput`), followed by its arguments in parentheses, which in this case are empty because this subroutine has no parameters.

Continuing the firefighter analogy, when firefighters arrive at the scene of the fire, they take control and maintain that control until they put out the fire. Similarly, once the subroutine is called, whether by user action or code, it takes control of the application, and no other code executes without being called by the subroutine, until the subroutine is finished. Thus, the line

```
Debug.WriteLine("When does this line display?")
```

in the `Click` event procedure of the form does not execute until the `PrintInput` subroutine is finished executing.

NOTE *An exception is that two methods may execute independently in a multithreaded application. Such an application is an advanced topic far beyond the introductory scope of this book.*

Completing the analogy, when the firefighters successfully put out the fire, they pack up their equipment and go back to the fire station, relinquishing control of the fire scene. Similarly, when the subroutine finishes executing, it relinquishes control of the application, and whatever code (or user action) follows the call of the subroutine determines the further flow of the application. Here, the line

```
Debug.WriteLine("When does this line display?")
```

in the `Click` event procedure of the form only executes after the `PrintInput` subroutine finishes executing.

Parameters

Returning to our firefighter analogy, when firefighters are called to a fire, they need to know the location of the fire, the type of fire (house fire, chemical fire, and so on) so they know what equipment to bring, and other pertinent information. The particular location and type of fire may well vary from call to call, but in each case this information is necessary in order for the firefighters to do their job.

Similarly, a procedure often needs information in order to perform its task. For example, a subroutine that outputs the square of a number to the Output window needs to know the number to be squared. The value of that number may vary from call to call, but in each case the procedure will need to know the particular number to be squared. This information is called a parameter.

If a procedure has no parameters, the parentheses following the procedure name are empty, and in fact are not required, though the IDE usually adds them. However, if a procedure has one or more parameters, each parameter must be declared within the parentheses.

The syntax for each parameter in the parameter list is as follows:

```
([ByVal|ByRef] parametername As datatype)
```

The keywords `ByVal` and `ByRef` will be discussed later in the section “`ByVal` vs. `ByRef`.” Next, `parametername` is a name, similar to a variable name, that will be used to refer to the parameter inside the body of the procedure. Finally, `datatype` is, as the word suggests, the data type of the parameter.

NOTE *As with variable names, you should name your parameters descriptively. For example, the parameter may be named “num” because that describes its purpose in the Power procedure, the number that will be raised to the second power.*

Passing an Argument to a Procedure

In the following example, the `Power` subroutine has one parameter, `num`, whose data type is an `Integer`. The body of the `Power` routine outputs the square of the value of its parameter to the Output window:

```
Private Sub Power(ByVal num As Integer)
    Debug.WriteLine( (num ^ 2).ToString )
End Sub
```

NOTE *The `ToString` method is used to convert an `Integer` into the `String` representation of an `Integer` because the `WriteLine` method expects an argument of the `String` data type.*

You could call the `Power` subroutine passing an `Integer` argument (for example, 5) from an event procedure such as the `Click` event of the form:

```
Private Sub Form1_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles Me.Click
    Power(5)
End Sub
```


Once again, calling a subroutine starts with the subroutine's name (here, `Power`), followed by its argument in parentheses (here, one argument because this subroutine was declared as having one parameter).

If you ran the program in debug mode and clicked the form, the value 25 would be displayed in the Output window.

The statement `Power(5)` has the effect of assigning the value 5 to the parameter `num` in `Power`. Therefore, the statement in `Power`

```
Debug.WriteLine( (num ^ 2).ToString )
```

executes as follows when the value passed is 5:

```
Debug.WriteLine( (5 ^ 2).ToString )
```

Multiple Parameters

A subroutine may require more than one argument in order to do its job. For example, the `Power` subroutine is limited to raising its argument to the power of 2. We can expand the functionality of this subroutine so it can raise the first argument to a different power than 2 by including a second argument, which is the power to which the first argument will be raised. The `Power` subroutine may be rewritten as follows:

```
Private Sub Power(ByVal num As Integer, _  
    ByVal exponent As Integer)  
    Debug.WriteLine(num ^ exponent)  
End Sub
```

As this example illustrates, the only difference between declaring a procedure with a single parameter and declaring a procedure with more than one parameter is that a comma separates the parameters. Similarly, when you call the procedure, a comma separates the arguments:

```
Power(5, 3)
```

The output with this call is 125, which is 5 to the power of 3.

The Parameters and Arguments Must Match

When you call a procedure, you must pass the same number of arguments as the number of parameters specified in the procedure's declaration. For example, let's return to our version of the `Power` subroutine with one parameter:

```
Private Sub Power(ByVal num As Integer)  
    Debug.WriteLine( (num ^ 2).ToString )  
End Sub
```

Suppose you tried to call the `Power` procedure with no arguments, as shown here:

```
Power()
```

In this case, the compiler would complain, “Argument not specified for the parameter ‘num’ of ‘Private Sub Power(num As Integer).’”

Similarly, suppose you tried to call the Power procedure with more than one argument, as shown here:

```
Power(5, 3)
```

This time, the compiler would complain, “Too many arguments to ‘Private Sub Power(num As Integer).’”

The argument passed also must be the same data type specified in the procedure’s declaration. For example, try to call the Power subroutine with code (presumably inside an event procedure) as follows:

```
Dim strInput As String  
strInput = InputBox("Enter a number")  
Power(strInput)
```

You are passing the correct number of arguments, one. Nevertheless, if Option Strict is on as I recommended in Chapter 5, the compiler would complain, “Option Strict On disallows implicit conversions from ‘String’ to ‘Integer.’” The reason is that the expected data type of the argument is an Integer, not a String. Therefore, the call would have to be rewritten as follows:

```
Power(Integer.Parse(strInput))
```

ByVal vs. ByRef

So far any arguments in the code examples have been preceded by the ByVal attribute. This section will illustrate the difference between the ByVal and ByRef attributes.

First, rewrite the Power function and the Click event procedure of the form to read as follows:

```
Private Sub Power(ByVal num As Integer)  
    Debug.WriteLine _  
        ("In Power num starts at " & num.ToString)  
    num = num ^ 2  
    Debug.WriteLine _  
        ("After num = num ^ 2 num = " & num.ToString)  
End Sub
```

```
Private Sub Form1_Click(ByVal sender As Object, _  
    ByVal e As System.EventArgs) Handles Me.Click  
    Dim x As Integer = 5  
    Debug.WriteLine _  
        ("Before calling Power x = " & x.ToString)
```

```
Power(x)
Debug.WriteLine _
    ("After calling Power x = " & x.ToString)
End Sub
```

Run the application. The output would be the following:

```
Before calling Power x = 5
In Power num starts at 5
After num = num ^ 2 num = 25
After calling Power x = 5
```

As the output demonstrates, although the value of the parameter `num` in the called procedure (`Power`) changed, the value of the corresponding argument `x` in the calling procedure (`Click` event of the form) did not. In other words, when a parameter has the `ByVal` attribute, any change to the value of the parameter in the called procedure does *not* affect the value of the corresponding argument in the calling procedure.

Stop the application and make one change to the code, changing the attribute of the parameter in the `Power` subroutine from `ByVal` to `ByRef`. The code now should read as follows:

```
Private Sub Form1_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles Me.Click
    Dim x As Integer = 5
    Debug.WriteLine _
        ("Before calling Power x = " & x.ToString)
    Power(x)
    Debug.WriteLine _
        ("After calling Power x = " & x.ToString)
End Sub

Private Sub Power(ByRef num As Integer)
    Debug.WriteLine _
        ("In Power num starts at " & num.ToString)
    num = num ^ 2
    Debug.WriteLine _
        ("After num = num ^ 2 num = " & num.ToString)
End Sub
```

Run the application. The output would be the following:

```
Before calling Power x = 5
In Power num starts at 5
After num = num ^ 2 num = 25
After calling Power x = 25
```

As the output demonstrates, when the value of the parameter `num` in the called procedure (`Power`) changed, the value of the corresponding argument `x` in the calling procedure (`Click` event of the form) also changed. In other words, when a parameter has the `ByRef` attribute, any change to the value of the parameter in the called procedure *does* affect the value of the corresponding argument in the calling procedure.

Normally you will use the `ByVal` attribute, which is the default. However, which attribute you use simply depends on what you are trying to accomplish.

Functions

As with subroutines, functions need to be created by declaring them, and then invoked by calling them. Also as with subroutines, you can pass information to functions by using arguments. However, unlike subroutines, functions present an additional consideration, the value they return.

Declaring Functions

The syntax for declaring a function is quite similar to that for declaring a subroutine:

```
[Accessibility] Function name[(Parameter list)] As Type
    Statements
    [Exit Function]
    Return [ or functionname = return value]
End Function
```

Table 9-3 lists and describes the elements of a function.

There is significant overlap between the syntax for declaring a function and the syntax for declaring a subroutine. The discussion for subroutines regarding accessibility, naming, argument lists, and statements applies equally to functions. Similarly, the `Exit Function` and `End Function` statements are to functions what the `Exit Sub` and `End Sub` statements are to subroutines. The sole difference is the substitution of the `Function` keyword for the `Sub` keyword.

The substantive differences in syntax between subroutines and functions, in addition to the use of the keyword `Function` rather than `Sub`, relate to the fundamental difference between a subroutine and a function—that a function, unlike a subroutine, returns a value. Here are two points to keep in mind:

- A function has As Type after the parameter list to describe the data type of the return value, whereas a subroutine does not because it does not return a value.
- In a function the keyword Return is followed by the value to be returned, or the function name is followed by an assignment operator and the value to be returned, whereas in a subroutine the keyword Return appears by itself because no value is being returned.

The following programmer-defined function, named ReturnInput, illustrates the syntax of a function. This function returns what the user enters in the input box,

Element	Required or Optional	Description
Accessibility	Optional	Determines where the function may be called from.
Function	Required	Keyword indicating the procedure is a function.
Name	Required	Name of the function.
Parameter list	Optional	Same as in a subroutine, the information passed to the function.
As Type	Required if Option Strict is On; otherwise, Optional	Data type of the return value.
Statements	Required	Code that executes each time the function is called.
Return (or functionname = return value)	Optional	Returns the value of the function. The Return statement also ends execution of the function before the End Function statement, whereas name = return value does not.
Exit Function	Optional	Ends execution of the function before the End Function statement.
End Function	Required	Ends execution of the function.

Table 9-3 Elements of a Function

unless the user did not enter anything or clicked the Cancel button of the input box, in which case the function returns the string “Nothing entered”:

```
Private Function ReturnInput() As String
    Dim strInput As String
    strInput = InputBox("Enter something")
    If strInput = "" Then
        Return "Nothing entered"
    End If
    ReturnInput = strInput
End Function
```

In the ReturnInput example, the As Type statement indicates that the data type of the return value is a String. The data type being returned may be any data type supported by .NET.

The Return statement executes if the value of strInput is an empty string, which would be the case if the user either did not enter anything or clicked the Cancel button of the input box. The Return keyword is followed by the value to be returned—in this case, the string “Nothing entered.”

If the Return statement executes, the execution of the function ends.

However, if the value of strInput is not an empty string (that is, the user entered something and clicked the OK button of the input box), then the execution of the function continues with the following statement:

```
ReturnInput = strInput
```

This statement also returns a value (here, that of the variable strInput). The syntax starts with the function name (ReturnInput), followed by the assignment operator and the value to be returned.

The section “How the Value Is Returned,” later in this chapter, will discuss further how a value is returned and the options in doing so.

Calling Functions

Calling a function is similar to calling a subroutine in that you refer to the function by name, followed by the arguments of the function in parentheses, or empty parentheses if the function has no arguments. The difference between calling a function and calling a subroutine concerns, once again, the return value of the function.

The most common scenario is to call the function on the right side of an assignment statement, with the left side of the assignment statement containing a variable or writable property of the same data type as the return value of the function. For example, converting the Power subroutine discussed earlier into a function, the

Power function declared as follows will return the result of raising its first argument to the value of its second argument:

```
Private Function Power(ByVal num As Integer, _  
    ByVal exponent As Integer) As Double  
    Return num ^ exponent  
End Function
```

You could call this function, such as in the Click event of the form, as follows, with the output being 125:

```
Private Sub Form1_Click(ByVal sender As Object, _  
    ByVal e As System.EventArgs) Handles Me.Click  
    Dim dblResult As Double  
    dblResult = Power(5, 3)  
    Debug.WriteLine( dblResult.ToString )  
End Sub
```

The steps are to declare a variable of the same data type as the return value of the function and then to assign the return value of the function to that variable.

You also can call the function in an expression so that the body of the Click event procedure of the form would be only one line:

```
Debug.WriteLine( Power(5, 3) )
```

The output still would be 125. The difference between calling the function on the right side of an assignment statement and calling the function in an expression is that when you call a function on the right side of an assignment statement, its return value is saved in a variable or writable property for later use. By contrast, when you call a function in an expression, its return value is not available for later use. Of course, this is not a problem unless you will need the return value later, which you may not.

NOTE *You do not have to use the return value of the function. You can call the function as you would call a subroutine, such as this: `Power(5, 3)`. In that event, the return value simply would not be used. However, whatever code was in the function body still would execute.*

How the Value Is Returned

The output of the preceding code is 125, but how exactly did that happen? Let's start with the call of the Power function by the following line:

```
dblResult = Power(5, 3)
```

The declaration of the Power function is shown here:

```
Private Function Power(ByVal num As Integer, _  
ByVal exponent As Integer) As Double  
    Return num ^ exponent  
End Function
```

Because the values of the arguments passed are 5 and 3, in that order, the value of the first parameter, num, is 5, and the value of the second parameter, exponent, is 3. Therefore, the statement

```
Return num ^ exponent
```

in effect is

```
Return 5 ^ 3
```

which, in turn, is

```
Return 125
```

With the Return statement, the Power function finishes executing, and the value 125 is returned to the right side of the assignment statement. After the Power function finishes executing, the statement

```
dblResult = Power(5, 3)
```

in effect is

```
dblResult = 125
```

Therefore, the following code outputs 125, the string representation of the value of dblResult:

```
Debug.WriteLine( dblResult.ToString )
```

Options When Returning a Value

As discussed in the earlier section “Declaring Functions,” there are two syntax options when returning a value: the Return statement, and assigning to the function name the value to be returned. Both methods are used in the earlier example of the ReturnInput function:

```
Private Function ReturnInput() As String  
    Dim strInput As String  
    strInput = InputBox("Enter something")  
    If strInput = "" Then  
        Return "Nothing entered"
```



```
End If
ReturnInput = strInput
End Function
```

The difference between the two alternatives is that the `Return` keyword immediately ends the execution of the function. In contrast, the syntax of returning a value by assigning it to the function name (for example, `ReturnInput = strInput`) does not end the execution of the function. The function continues executing until a `Return`, `Exit Function`, or `End Function` statement is reached, and the return value assigned to the function name will remain the function's return value unless changed, either by assigning a different return value to the function name or by using the `Return` statement. You may use to your advantage that assigning a return value to the function name does not end the execution of the function by assigning a preliminary return value and adjusting it later in the function if necessary.

NOTE *Technically, a function does not have to explicitly return a value. If the function ends, either by an `Exit Function` or `End Function` statement, without previously returning a value by either the `Return` statement or assigning a value to the function name, then the function returns the default value appropriate to the data type of the return value. This is 0 for numeric data types; Nothing for Object, String, and all arrays; and False for Boolean. However, it is good practice for your functions to return a value.*

Returning a Boolean Value

Functions that return a Boolean value often are called in an expression in an `If...Then` control structure. For example, the following function, `IsEmptyString`, returns `True` if the string that is its argument is an empty string, and otherwise returns `False`:

```
Private Function IsEmptyString _
    (ByVal str As String) As Boolean
    If str = "" Then
        Return True
    Else
        Return False
    End If
End Function
```

The function may then be called following an `If` clause, and passed a string input by the user in an input box. If the user did not enter anything in the input box or clicked the `Cancel` button, the function will return `True` and the output will be, "You

didn't enter anything." If instead the user entered something in the input box and clicked the OK button, the function will return False and the output will be, "You entered something."

```
Private Sub Form1_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles Me.Click
    Dim strInput As String
    strInput = InputBox("Enter something")
    If IsEmptyString(strInput) Then
        Debug.WriteLine("You didn't enter anything")
    Else
        Debug.WriteLine("You entered something")
    End If
End Sub
```

The statement

```
If IsEmptyString(strInput) Then
```

also could have been written as follows:

```
If IsEmptyString(strInput) = True Then
```

The two statements have the same effect. Because `IsEmptyString` returns a Boolean value, it is unnecessary to compare that Boolean value to another Boolean value to obtain a Boolean result. Therefore, the `= True` is unnecessary, though harmless.

Why Write Your Own Procedures?

This chapter has explained how you can write your own procedures. However, it has not yet explained why you would want to do so. Indeed, often you could write all your code inside event procedures and never have to write your own procedures.

By analogy, assume this book is a Visual Basic application, and each chapter is an event procedure. Each chapter is divided into headings and subheadings to make the chapter more readable. The headings and subheadings, which I would analogize to programmer-defined procedures, are not necessary, but without them, each chapter, averaging about 15 to 20 pages long, would be more difficult to read and understand.

Similarly, as your applications become more complex, the code in your event procedures may become very long if not separated and organized into programmer-defined procedures. Although lengthy code will run fine, it is more difficult to understand and, if necessary, fix than code that is organized into brief code blocks.

Additionally, if you are performing essentially the same task from several places in the program, you can avoid duplication of code by putting the code that performs that task in one procedure, as opposed to repeating that code in each place in the program that may call for the performance of that task. That way, if you later have to fix a bug in how you perform that task, or simply find a better way to perform the task, you only have to change the code in one place rather than many.

Conclusion

A procedure is a block of one or more code statements that execute when called upon to do so by an event or code. Most Visual Basic code consists of procedures.

This chapter discussed two types of procedures: subroutines and functions. The difference between them is that functions return a value, whereas subroutines do not.

Visual Basic has many built-in procedures. Some, such as event procedures, are subroutines. Others, such as the `InputBox` function, are functions.

Visual Basic also enables you to create your own procedures. There are several reasons why you might want to create your own procedures. Your code is more readable if divided up among several smaller procedures than all contained in one procedure that contains pages of code. Additionally, if you are performing essentially the same task from several places in the program, you can avoid duplication of code by putting the code that performs that task in one place, as opposed to repeating that code in each place in the program that may call for the performance of that task. Further, if you later have to fix a bug in how you perform that task, or simply find a better way to perform the task, you only have to change the code in one place rather than many.

You also learned in this chapter how to pass information to a procedure using arguments, and how to call a procedure so the code within it will execute. You also learned how to return a value from a function, and how to assign that return value when you call the function.

So far our applications have involved only one form. That will change in the next chapter.

Quiz

1. What is a procedure?
2. What is the difference between a subroutine and a function?
3. Is an event procedure a subroutine or a function?
4. What does the Private access specifier do when applied to a procedure?
5. Is there a difference between the Return and Exit Sub statements in subroutines?
6. What does calling a subroutine do?
7. What is the difference between the ByVal and ByRef attributes of a parameter?
8. What is the difference between a subroutine and a function in the use of the keyword Return?
9. What are the two syntax options for a function returning a value?
10. What are some reasons for writing your own procedures?



PART FOUR

The User Interface

This page intentionally left blank

Helper Forms

Forms are the most common user interface element in Visual Basic applications. Indeed, it is difficult to conceptualize a Windows application without at least one form. Forms are the windows, literally, through which application users view information and interact with the application.

Visual Basic's automated creation of a new Windows application project includes a form that serves as the main application window. However, although the main application window may be the star of the show, that form needs a supporting cast of helper forms, because Windows applications generally are far too complex for the main application window to perform all the tasks required by the application.

The message box is a helper form built into the .NET Framework. The message box includes text that is either informative or a question, as well as buttons such as OK, Yes, No, Cancel, and so on, for the application user's response and to close the message box.

Message boxes are very common in Windows applications. One typical example, discussed later in this chapter, is if you make changes to a document in Microsoft Word and then try to close the document without saving the changes, you may be presented with a message box asking if you want to save the file before closing, with buttons for Yes, No, and Cancel. This chapter will show you how to create and use a message box in your application.

Although the message box is very useful, sometimes you want the helper form to have functionality that is beyond the capability of a message box to provide. For

example, the text displayed by a message box is limited to a prompt. However, most Windows applications have an About dialog box, summoned by the main form's Help | About menu command, that displays more detailed information about the application than can be provided in a message box.

The About dialog box is an example of a dialog form. However, although the About dialog box simply is informational, dialog forms are not limited to the role of passive purveyors of information, and instead typically are interactive. For example, the Print dialog box, displayed with the File | Print menu command, enables the user to choose among printers, decide which pages to print, the number of copies to make, and so forth, and then starts the print job when the OK button is clicked. This chapter will show you how to create and display a dialog form.

The ability of the user to interact with the Print dialog box is possible because that dialog box contains controls that a message box cannot contain, such as a drop-down list from which the user may select a printer, radio buttons and a text box from which the user may designate which pages to print, a check box through which the user can designate whether the pages should be collated, and so forth.

The ability of the user to interact with a dialog form presents programming challenges involving communication between the main and helper forms. For example, the main form needs to know which button was clicked on the helper form, and should execute different code depending on which button was clicked. Additionally, because the dialog form contains controls, the main form needs to know and take actions based on what the application user typed, checked, or selected in the controls in the helper form. This chapter will show you how to solve these programming challenges.

Message Boxes

Because the actions of the application user cause a Windows application to receive messages from the operating system, it seems only fair that a Windows application can send a message to the application user. Windows applications often use message boxes to inform and obtain a response from the application user.

Message boxes are valuable tools to use in applications. For example, one late evening, working bleary-eyed to finish a chapter under unceasing pressure from my heartless editor, I forgetfully closed the document without first saving about an hour's worth of changes. Mercifully, up popped the message box shown in Figure 10-1, asking if I wanted to save my unsaved changes before the document was closed.

This message box, in addition to conveying valuable information, also is able to obtain and process my response. If I choose the Yes button, the unsaved changes are saved before the document is closed. If I choose the No button (bad choice), the unsaved changes are discarded and the document is closed. If I choose the Cancel button, the state just before I attempted to close the document is restored; the document is kept open, but the unsaved changes remain unsaved.



Figure 10-1 Message box in Microsoft Word

Creating the Project

In this project, you will create the message box shown in Figure 10-2, which asks the user if they want to quit the application. If the user chooses Yes, the application closes. If the user chooses No, the application will not close.

Create the project through the following steps:

1. Create a new Windows application.
2. Using the Toolbox, add a button to the form.
3. Use the Properties window to change the Name property of the button to `btnClose` and the Text property of the button to Close.
4. Add this code to the Click event of `btnClose`:

```
Private Sub btnClose_Click _  
    (ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles btnClose.Click  
    Dim drQuit As DialogResult  
    drQuit = MessageBox.Show _  
        ("Do you really want to quit?", _  
        "Exit Confirmation", _  
        MessageBoxButtons.YesNo, _  
        MessageBoxIcon.Warning, _  
        MessageBoxDefaultButton.Button2)  
    If drQuit = DialogResult.Yes Then  
        Me.Close()  
    End If  
End Sub
```

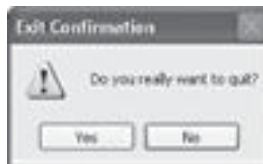


Figure 10-2 Project in action

Run the project and click the Close button to display the message box shown previously in Figure 10-2. This type of message box is common in Windows applications, providing the application user a last chance to decide whether they really want to quit the application. If the application user chooses the Yes button, the application will end. If instead the application user chooses the No button, just the message box will close and the application user will be returned to the main form. Thus, the clicking of the No button will restore the application to its state just before the application user chose the Close button.

Message Boxes Are Modal

The code involves three logical steps:

1. Display the message box, using the Show method.
2. Obtain the application user's choice, Yes or No, by the return value of the Show method.
3. If the choice is Yes, close the application.

However, before we analyze the code, let's examine a feature that message boxes share with the dialog forms discussed later in this chapter—both are modal.

The term “modal” refers to the fact that the user cannot return to the application until the message box is closed by the user clicking one of the buttons of the message box.

Message boxes are always modal. However, not all forms are modal. This issue will be discussed further in connection with dialog forms later in this chapter in the section “Modal vs. Modeless.”

Show Method

You do not need to create or design the message box. The message box is a form built into the .NET Framework. All you need to do to create and display a message box, together with its buttons, icon, text, and title, is to call the aptly named Show method of the MessageBox class, which is part of the class library of the .NET Framework, and provide the appropriate arguments. The .NET Framework also takes care of closing the message box. When you click a button, the message box closes, automatically.

Parameters of Show Method

The Show method is overloaded. This means that you can call it several different ways, depending on the number of parameters you include. The parameters of the Show method are listed in Table 10-1.

Parameter	Description	Required?
Text	The prompt inside the message box to convey a question or information to the application user (in this case, “Do you really want to quit?”).	Yes
Title	The title of the message box (in this case, “Exit confirmation”) to provide a visual cue to the application user of the purpose of the message box.	No. If omitted, no title.
MessageBoxButtons	The buttons inside the message box (in this case, Yes and No). The choices are listed in Table 10-2.	No. If omitted, only one button (OK).
MessageBoxIcon	The graphic inside the message box, such as the ! in Figure 10-2. The choices are listed in Table 10-3.	No. If omitted, no graphic.
MessageBoxDefaultButton	The button outlined as a cue that pressing ENTER is the same as clicking the button (in this case, the second, or No button). The choices are listed in Table 10-4.	No. If omitted, first button is the default.

Table 10-1 Parameters of the Show Method

The only parameter that is required is Text. In that case, the message box only will have one button, OK, which closes the message box when clicked. This may be sufficient if the message box simply provides information to the application user. For example, when filling out a form in an application, you may have seen a message box popping up telling you that you forgot to fill out a required field, or that the field only takes numbers or that the password must be at least six characters, and so on.

NOTE *The parameters are positional. This means you can’t skip or omit an argument. Therefore, if you want to specify a default button, which is the last parameter, all the previous arguments must also be supplied.*

MessageBoxButtons Enumeration

Although a message box with only an OK button is sufficient if the message box's purpose is purely information, here the objective of this project is to give the application user a choice of Yes or No concerning whether they really want to quit. You use buttons—here, Yes and No buttons—to give the application user this choice. The MessageBoxButtons enumeration contains the available button combinations, which are listed in Table 10-2.

The term “enumeration” means a list of related choices, which in this case represents the various available button combinations. The syntax of an enumeration is

```
[Enumeration Name].[Choice Name]
```

For example, if the selected button combination is Yes and No, the syntax is

```
MessageBoxButtons.YesNo
```

Here, MessageBoxButtons is the name of the enumeration, and YesNo is the choice from the enumerated list.

MessageBoxIcon Enumeration

The saying that a picture is worth a thousand words, while perhaps trite, has much truth. The visual cue of an icon in a message box tells the application user the nature and importance of the message, ranging from informational to warning or error.

Similar to the buttons, the available icon choices are contained in an enumeration, this time named the MessageBoxIcon enumeration. Table 10-3 lists the available icon choices.

Name	Buttons Contained in Message Box
AbortRetryIgnore	Abort, Retry, and Ignore.
OK	OK. This is the default.
OKCancel	OK and Cancel.
RetryCancel	Retry and Cancel.
YesNo	Yes and No.
YesNoCancel	Yes, No, and Cancel.

Table 10-2 MessageBoxButtons Enumeration

Name	Icon in Message Box
Asterisk	White lowercase letter i in a circle with a blue background
Error	White X in a circle with a red background
Exclamation	Black exclamation point in a triangle with a yellow background
Hand	White X in a circle with a red background
Information	White lowercase letter i in a circle with a blue background
None	None
Question	Blue question mark in a circle with a white background
Stop	White X in a circle with a red background
Warning	Black exclamation point in a triangle with a yellow background

Table 10-3 MessageBoxIcon Enumeration

MessageBoxDefaultButton Enumeration

The users of your application may be using the keyboard in lieu of the mouse to choose a button. This may not simply be a matter of preference. Users with certain disabilities may not be able to use a mouse and have to use the keyboard to choose a button. Accordingly, you should designate a default button, which means that the user pressing the `ENTER` key is the same as the user clicking that button.

The choices of the default button are contained in yet another enumeration, this time called the `MessageBoxDefaultButton` enumeration. Table 10-4 lists the available button choices.

Member Name	Description
Button1	The first button on the message box is the default button.
Button2	The second button on the message box is the default button.
Button3	The third button on the message box is the default button.

Table 10-4 MessageBoxDefaultButton Enumeration

There are only three buttons in the enumeration because, as Table 10-2 indicates, the maximum number of buttons is three—Abort, Retry and Ignore, or Yes, No, and Cancel.

Usually you choose as the default button the one whose choice would have the least drastic effect, if for no other reason than if the application user absentmindedly presses the ENTER key, nothing horrible will happen. Here, the button with the least drastic effect is the No button, which will simply restore the status quo.

Using the Show Method's Return Value

The next step is to write code so the form knows if the application user clicked the Yes or No button in the message box. The programming task is that one form needs to know an action taken in another form, the other form here being the message box.

You solve this problem by using the return value of the Show method. The concept of a procedure returning a value was introduced with the InputBox function and discussed further in Chapter 9 in the coverage of functions.

DialogResult Enumeration

The Show method returns a value that represents the button that the application user clicked in the message box. Each button is represented by a member of the DialogResult enumeration listed in Table 10-5.

Member Name	Description
Abort	The dialog box's return value is Abort, usually sent from a button labeled Abort.
Cancel	The dialog box's return value is Cancel, usually sent from a button labeled Cancel.
Ignore	The dialog box's return value is Ignore, usually sent from a button labeled Ignore.
No	The dialog box's return value is No, usually sent from a button labeled No.
None	Nothing is returned from the dialog box. This means that the modal dialog box continues running.
OK	The dialog box's return value is OK, usually sent from a button labeled OK.
Retry	The dialog box's return value is Retry, usually sent from a button labeled Retry.
Yes	The dialog box's return value is Yes, usually sent from a button labeled Yes.

Table 10-5 DialogResult Enumeration

The `DialogResult` enumeration corresponds to the buttons in the `MessageBoxButtons` enumeration listed previously in Table 10-2, and will be returned if the corresponding button is chosen. Thus, if the application user chooses the Yes button, the `Show` method returns the value `DialogResult.Yes`.

The return value usually is stored in a variable for later use in the application. The data type of that return value should be the same as the data type returned by the function or method.

Accordingly, you often use the `DialogResult` data type for the variable in which you will save the return value of the `Show` method. You may declare that variable as follows:

```
Dim drQuit As DialogResult
```

Once you have declared the variable, the next step is to use it to store the return value of the `Show` method. The variable `drQuit` should be on the left side of the assignment operator, so it will receive the return value of the `Show` method that is called on the right side of the assignment operator:

```
drQuit = MessageBox.Show("Do you really want to quit?", _  
    "Exit Confirmation", _  
    MessageBoxButtons.YesNo, _  
    MessageBoxIcon.Warning, _  
    MessageBoxDefaultButton.Button2)
```

When this code statement executes, and the application user clicks a button in the message box, closing the message box, the value of the variable `drQuit` will be either `DialogResult.Yes` or `DialogResult.No`, depending on whether the application user clicked the Yes or No button.

Processing the Returned DialogResult Value

The form object has a `Close` method that, as its name indicates, closes the form. Because this is the only form in the project (other than the message box, which will close when the user clicks the Yes or No button), closing the form ends the application as well. However, we only want to close the form if the application user clicks Yes, not if the application user clicks No.

The following code closes the form if, and only if, the application user's choice was Yes:

```
If drQuit = DialogResult.Yes Then  
    Me.Close()  
End If
```

This code statement first compares the value of `drQuit` and `DialogResult.Yes` using the `If` keyword. If the user chose Yes, the value of `drQuit` is `DialogResult.Yes`,

so the comparison `drQuit = DialogResult.Yes` will be `True` and the `Me.Close()` statement is executed. However, if the user chose `No`, the value of `drQuit` is `DialogResult.No`, so the comparison `drQuit = DialogResult.Yes` will be `False` and the `Me.Close()` statement will not be executed.

Dialog Forms

Although the message box is a valuable tool, it is limited in that it only can contain a text prompt, buttons, an icon, and a title. Further, the only information a message box can obtain from the application user is which button the user clicked. The message box does not permit the application user to enter text in a text box, choose an item from a drop-down list, select a check box or radio button, and so on.

If you need a user interface richer than the message box, you may create a custom and more complex version of a message box—the dialog form.

Creating the Project

A good way to illustrate how to create and use a dialog form is with a project. In this project, you will create the dialog form shown in Figure 10-3. This dialog form enables the user to change the text of the title bar of the main form, that title bar text currently being “Form1” in Figure 10-3.



Figure 10-3 Dialog Form project in action

Clicking either the OK or Cancel button will close the dialog form. However, if the user chooses the OK button in the dialog form, the text of the title bar of the main form will be changed to the text the user typed into the text box of the dialog form. By contrast, if the user instead chooses the Cancel button in the dialog form, the dialog form simply will close, with no change made to the text of the title bar of the main form.

Try the following steps to create this project:

1. Create a new Windows application.
2. Using the Properties window, change the StartPosition property of the form from the default (WindowsDefaultLocation) to CenterScreen to center the form on the screen. This change is not required for the program to function, but will permit both forms to be centered on the screen.
3. Using the Toolbox, add a button to the form.
4. Use the Properties window to change the values of the Name property of the button to btnNewCaption and the Text property from the default (such as Button1) to New Caption.
5. You need to add a second form to the project to serve as the dialog form. Use the Project | Add Windows Form menu command to display the Add New Item dialog box shown in Figure 10-4, highlight Windows Form, and then click the Add button. You can keep the default name, Form2.vb, for the new form. Figure 10-5 shows the Solution Explorer, in which the second form now appears.

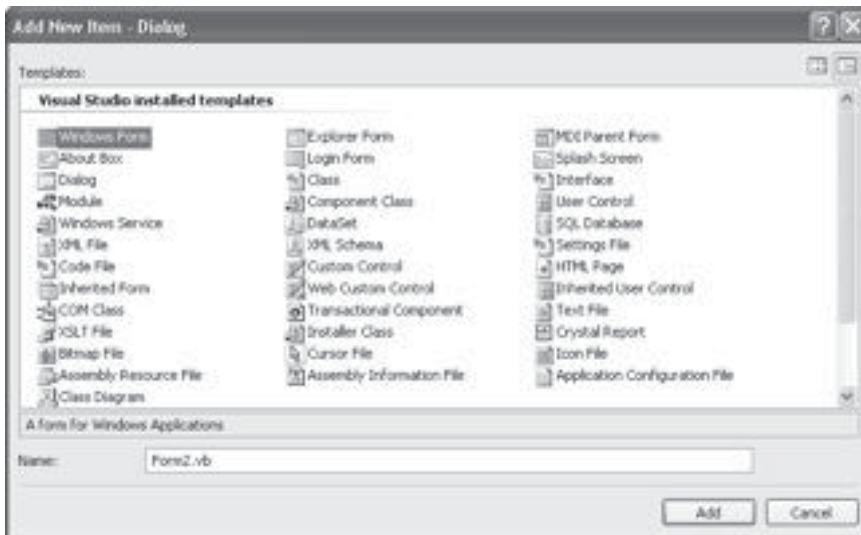


Figure 10-4 Add New Item dialog box

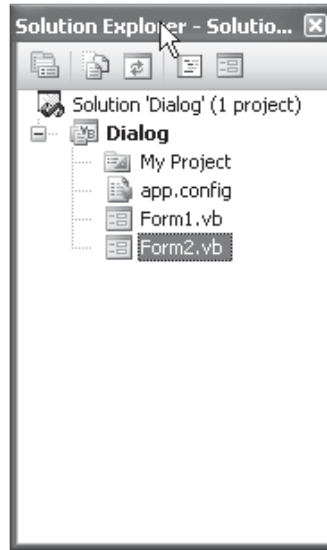


Figure 10-5 Solution Explorer after the second form is added

6. Using the Properties window, change the values of the following properties of the second form:
 - **Text** Change from Form2 to Dialog so you have a visual cue that you are looking at the dialog form.
 - **ControlBox** Change from the default (True) to False. This eliminates the close, minimize, and maximize buttons in the top-right corner of the window and the system menu, which also has close, minimize, and maximize commands, in the top-left corner of the window. The purpose is so the dialog form cannot be resized and can be closed only by clicking one of the buttons that you will be adding next to the form.
 - **StartPosition** Change from the default (WindowsDefaultLocation) to CenterParent so the dialog box is centered on the main form.
 - **FormBorderStyle** Change from the default (Sizable) to FixedDialog. This change is not required for the program to function, but does give the form a more dialog box–like appearance.
7. Using the Toolbox, add a button to the second form.

8. Use the Properties window to change the values of the following properties of the button you just added to the dialog form:
 - **Name** Change from Button1 to btnOK.
 - **Text** Change to OK.
 - **DialogResult** Choose OK from the drop-down list. Because the dialog form displayed by the `MessageBox.Show` method is a built-in Visual Basic .NET form, clicking the OK button automatically returns OK as the `DialogResult` value. By contrast, the dialog form is not a built-in Visual Basic .NET form, but instead one that you create, so you need to correlate the clicking of the OK button with OK as the `DialogResult` value, both in order to return a `DialogResult` value and also to close the dialog form when the button is clicked. You do so by setting the button's `DialogResult` property to OK.
9. Using the Toolbox, add a second button to the dialog form.
10. Use the Properties window to change the values of the following properties of the second button you just added to the dialog form:
 - **Name** Change from the default name (likely Button1 or Button2) to btnCancel.
 - **Text** Change to Cancel.
 - **DialogResult** Choose Cancel from the drop-down list. This is done for the same reason as when we set the `DialogResult` property of the OK button to OK.
11. Use the Properties window to change the values of the `AcceptButton` property of the second dialog form to btnOK and the `CancelButton` property of that form to btnCancel, using the drop-down list. Pressing the `ENTER` key is the equivalent of clicking the button designated in the `AcceptButton` property. Similarly, pressing the `ESC` key is the equivalent of clicking the button designated in the `CancelButton` property.
12. Using the Toolbox, add a `TextBox` control to the second form.
13. Use the Properties window to change the values of the following properties of the `TextBox` control you just added to the dialog form:
 - **Name** Change to txtNewCaption.
 - **Text** Delete the default so it is blank. This way, no text shows in the text box when you run the application.
 - **TabIndex** Change to 0 so when the second form appears the cursor will start at the text box.

14. Add the following code to the Click event of btnNewCaption in the main form:

```
Private Sub btnNewCaption_Click _  
    (ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) _  
    Handles btnNewCaption.Click  
    Dim frmCaption As New Form2()  
    frmCaption.ShowDialog(Me)  
    If frmCaption.DialogResult = DialogResult.OK Then  
        Me.Text = frmCaption.txtNewCaption.Text  
    End If  
End Sub
```

Try out this code by running the project. Click the New Caption button in the first form and then type some text in the second form. If you then click OK, the second form will close, and the first form will have a new title, the text you typed in the second form. If you instead click Cancel, the second form will still close, but the title of the first form will not change.

Showing the Dialog Form and Returning Its Result

The dialog form is similar to the `MessageBox` class. For example, both are displayed by another form and both are modal; that is, the application user cannot return to the main form until they have dismissed the dialog form by clicking one of its buttons. Another similarity is that both the dialog form and the `MessageBox` class return a result based on which button was clicked. However, there are important differences between the dialog form and a message box, both in how they are shown and in how they return a result.

ShowDialog Method

You use the `ShowDialog` method of the `Form` object to display a dialog form. This method is similar to the `Show` method of the `MessageBox` class in that it will show, modally, the form that is invoking the method.

NOTE *You also could display the second form using the `Show` method instead of the `ShowDialog` method, but then the second form would not be modal. This is discussed further in the later section “Modal vs. Modeless.”*

Because `Form2` is a class (that is, a blueprint or template for an object), the code first declares and creates an instance of `Form2` before you show it using the

ShowDialog method. You do so by the following code, which goes in the Click event procedure of the btnNewCaption button in the main form:

```
Dim frmCaption As New Form2()  
frmCaption.ShowDialog(Me)
```

Let's go through this code one line at a time.

The first line creates an object named frmCaption of the Form2 class. You use a class to instantiate (create) an object of that class. The class in this example is Form2. The New keyword is used to create the object. The object is represented by a variable (here, frmCaption).

The second line of code displays the dialog form object created in the first line. The Form2 object, represented by the variable frmCaption, calls the ShowDialog method to display itself as a dialog form. The Me keyword is passed as the argument. The Me keyword refers to the current form, which is the main form because we are writing this code in the main form. This makes the current, main form instance the owner of the dialog form.

Returning a DialogResult

Another difference between the MessageBox class and the dialog form is that whereas the Show method of the MessageBox class indicates the button the user clicked by returning a DialogResult value, the ShowDialog method of the Form object indicates the button the user clicked by assigning that value to the dialog form's DialogResult property. Thus, the comparison is

```
If frmCaption.DialogResult = DialogResult.OK Then
```

You can make multiple comparisons. For example, if the dialog form had three buttons, Yes, No and Cancel, the comparison could be the following:

```
If frmCaption.DialogResult = DialogResult.Yes Then  
    ' do action based on user clicking yes button  
ElseIf frmCaption.DialogResult = DialogResult.No Then  
    ' do action based on user clicking no button  
Else  
    ' do action based on user clicking cancel button  
End If
```

If DialogResult is anything besides None, the dialog form is closed and a DialogResult value is returned. However, under certain circumstances you may wish to prevent the dialog form from being closed, such as if the user has made an input error that first needs to be corrected.

To prevent the dialog form from closing, the DialogResult property of the dialog form needs to be set to None. The following code fragment sets the value of

the DialogResult property of the current form (represented by the Me keyword) to a DialogResult of None:

```
Me.DialogResult = DialogResult.None
```

This code logically would be placed in the Click event of the OK button to handle the situation where you want the user to fix an error on that dialog form rather than closing the dialog form.

Accessing Values from the Dialog Form

If the value of the second form's DialogResult property is OK, all that is left to do is to change the title of the first form to the text you typed in the second form. The following code in the Click event procedure of btnNewCaption therefore is indicated:

```
Me.Text = frmCaption.txtNewCaption.Text
```

The Me keyword refers to the main form because this code is in its code module. The Text property is the text in its title bar.

The reference to txtNewCaption, the text box in the dialog form, is preceded by the name of the dialog form object, frmCaption. The reason why the name of the control is preceded by the name of the form that contains it is that a reference to a control, not preceded by a form object, is assumed to be to a control in the form whose code is executing. However, the current code module is for the main form, and txtNewCaption is not in that form, but instead in the dialog form. Therefore, a reference to txtNewCaption.Text instead of frmCaption.txtNewCaption.Text would result in the following compiler error message: "The name 'txtNewCaption' is not declared."

Modal vs. Modeless

Although all message boxes are modal, not all forms are. The second form in the application we just created is a dialog form because it was displayed with the ShowDialog method rather than the Show method. Had we instead displayed the second form using the Show method, the second form would have been modeless. This means that the application user could return to the main form without closing the second form.

Some forms in Windows applications are modeless. Examples include the Find and Replace forms in Microsoft Word. Because the Find form is modeless, you can return to the main application window and edit a found word without having to close the Find form.

It usually is easier to write code for modal forms because you don't have to be concerned about the user returning to the main application without first closing the modal form. However, there are situations, such as the Find form in Microsoft Word, in which a modeless form may be the better choice.

Conclusion

Visual Basic 2005's automated creation of a new Windows application project includes a form that serves as the main application window. The main application window often needs a supporting cast of other forms, because Windows applications generally are far too complex for the main application window to perform all the tasks required by the application.

This chapter first showed you how to display a message box and determine which button the user clicked. You also learned that a message box is modal, which means that the user cannot return to the rest of the application until the message box is closed, by clicking one of its buttons.

You next learned how to create and use a dialog form. The dialog form is similar to the `MessageBox` class in that it is modal and returns a value based on the button clicked to dismiss it. However, a dialog form, unlike a message box, also may contain text boxes, check boxes, drop-down lists, and other controls.

There also are code differences between the dialog form and the message box. You use the `ShowDialog` method instead of the `Show` method to display a dialog form. Further, you first create an instance of the dialog form to use the `ShowDialog` method. Additionally, the return value of the `MessageBox` class is a `DialogResult` value, whereas the return value of the dialog form is in its `DialogResult` property. You also learned how, through code in the main form, to determine values in controls in the dialog form.

In the next chapter we will enhance the user interface of the form with a menu.

Quiz

1. Is a message box modal or modeless?
2. What value is returned by the `Show` method of the `MessageBox` class?
3. Do you always have to call the `Show` method of the `MessageBox` class with the same number of arguments?

4. Do buttons in a message box automatically have a DialogResult value?
5. What is the data type of a variable you may use to store the return value of the Show method of the MessageBox class?
6. What is an enumeration?
7. What method do you use to display a modal form?
8. What is the return value from showing a dialog form?
9. Do buttons in a dialog form you create automatically have a DialogResult value?
10. What method do you use to display a form as modeless rather than modal?

Menus

You often may encounter menus, perhaps at an elegant restaurant, or in my case, in the drive through lane of a local fast food restaurant. Regardless of the quality of the food, the menus at the two places serve the same purpose: to inform you of your choices and the corresponding prices.

A Windows application also has a menu, but that menu serves a different purpose than a restaurant menu. The application user generally knows what they want to do. The menu provides a graphical user interface (GUI) to make it easier for the application user to issue commands to the application, such as to open a file, print a document, and so on.

The menu is not the only way through which the GUI may make it easier for the application user to issue commands to the application. For example, toolbars, which are covered in the next chapter, are another alternative. However, the menu has the advantage of enabling the programmer to organize commands in a logical hierarchy. For example, commands related to file operations, such as New, Open, and Save, are under the File menu, whereas commands related to editing, such as Cut, Copy, and Paste, are under the Edit menu. Additionally, menus save valuable screen space, in that submenu items collapse unless the menu item above them is chosen. This enables your application to remain uncluttered, by hiding commands that are not immediately needed.

There are two common types of menus. One is the main menu that usually appears at the top of applications, with headings such as File, Edit, View, and Help. The main menu is represented by the `MenuStrip` class. The other menu that appears when you right-click, sometimes called a *shortcut* or *context menu*, is represented by the `ContextMenuStrip` class.

This chapter will show you how to create a main menu and a context menu, and link them to each other.

Creating a Main Menu

The `MenuStrip` class represents the main menu that usually appears at the top of a Windows form. The `MenuStrip` object contains a collection of `ToolStripMenuItem` objects, each of which is an item on the menu.

Each `ToolStripMenuItem` can be a command for your application. Figure 11-1 shows menu items under the File menu in Microsoft Word. Many of the menu items are commands for the application, such as to open or save a file.

However, as Figure 11-2 shows, a menu item may also be a parent menu for other menu items, each another `ToolStripMenuItem`. For example, Send To is the

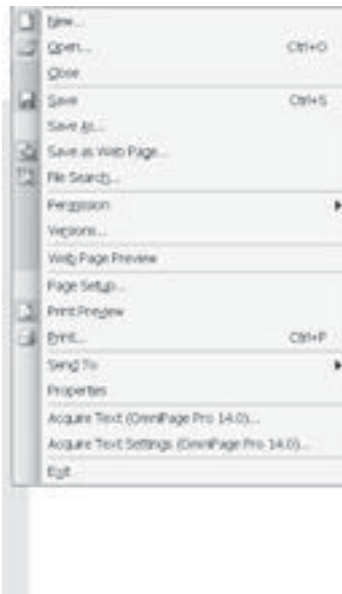


Figure 11-1 Menu items under the File menu

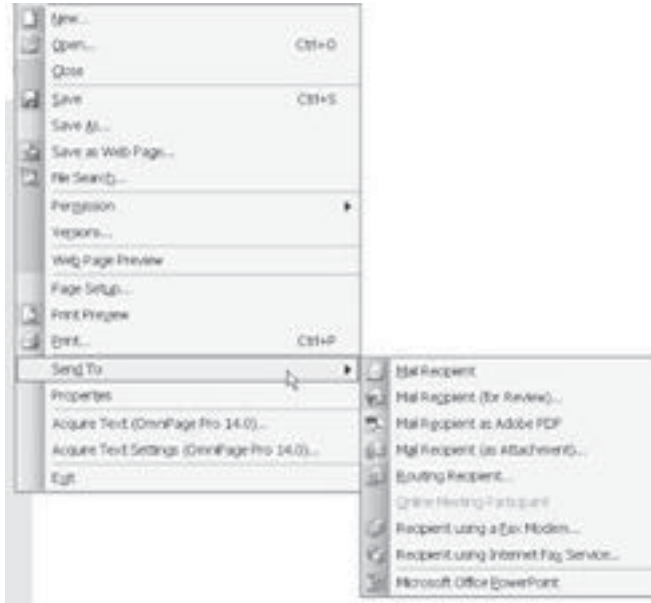


Figure 11-2 Send To menu item as a parent to other menu items

parent menu item for other menu items, including Mail Recipient and Microsoft Office PowerPoint.

Creating a main menu is a two-step process. You first add a MenuStrip control to your form, and then you append ToolStripMenuItem objects to it.

Adding a MenuStrip Control to a Form

You add a MenuStrip control to a form using the following steps, which are similar to how you would add a control such as a Button to the form. Try the following, which you could do with an existing project, though I would recommend a new project to avoid any confusion with existing code:

1. View the form in designer view.
2. Double-click the MenuStrip component in the Toolbox. As shown in Figure 11-3, the MenuStrip component is added to the component tray below the form. When this component is selected in the component tray, a rectangular area appears underneath the top-left corner of the form displaying the text “Type Here.”

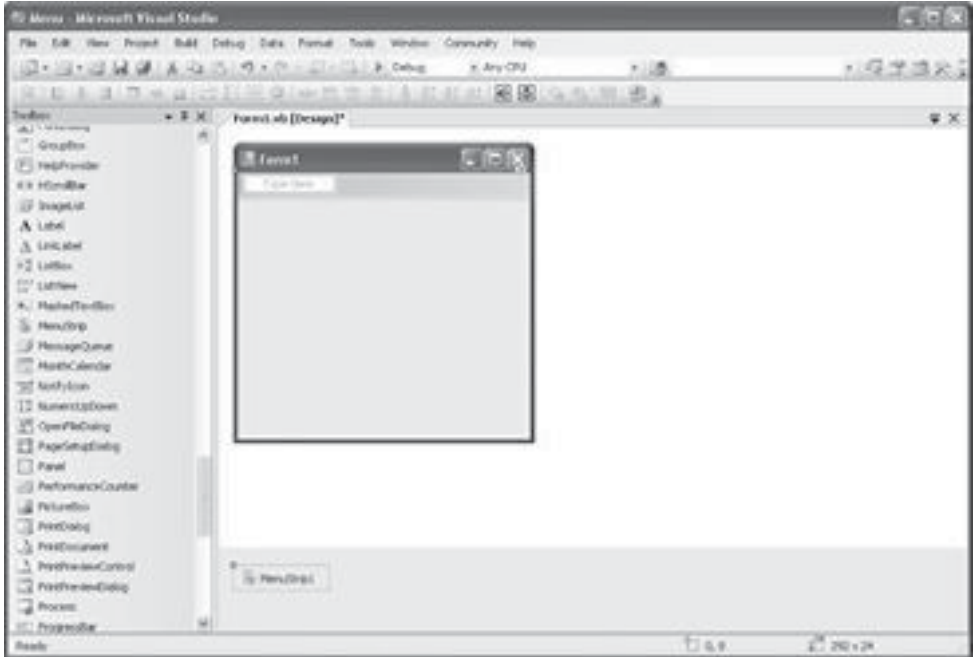


Figure 11-3 MenuStrip added to form

3. Using the Properties window, if not already set by default, set the `MainMenuStrip` property of the form to the name of your `MenuStrip` component (by default `MenuStrip1`). This links the `MenuStrip` to your form.

Adding Menu Items to the MenuStrip

Once you have added a `MenuStrip` component to your Windows form, the next step is to add menu items to it. Each menu item is an object of the `ToolStripMenuItem` class. You can add `ToolStripMenuItems` to the `MenuStrip` by typing in the menu items or by using the Items Collection Editor.

Typing in the Menu Items

You may add a menu item to the `MenuStrip` component by clicking the text “Type Here” (after selecting the `MenuStrip` component in the component tray as mentioned in step 2 in the preceding section) and typing the display name of the desired menu item to add it. For example, you may add a File menu item by typing **File**—the File menu usually is the first top-level item in Windows applications.

Typing the name of the menu item sets its Text property. You also should change the menu item's Name property from the default. You set the Name property of the menu item by right-clicking it, choosing Properties from the shortcut menu to display the Properties window, and then changing the Name property in the Properties window. One logical name for the File menu would be `mnuFile`, with the "mnu" prefix indicating a menu item and "File" indicating the purpose of the menu item.

Figure 11-4 shows the menu after the File menu item is added.

As Figure 11-4 shows, you now have "Type Here" options both below and to the right of the File menu item. You may add items below the File menu item, such as New and Open. You then should change the Name property of these menu items. For example, I would name the menu item Open under the File menu `mnuFileOpen`, with "mnuFile" being the name of the parent File menu and "Open" being descriptive of the subsidiary menu item's purpose.

You may add menu items to the right of the File menu as well as below it. For example, you might add an Edit menu item to the right of the File menu item to be consistent with other Windows applications. Following the same naming convention, I would name the Edit menu item `mnuEdit`.

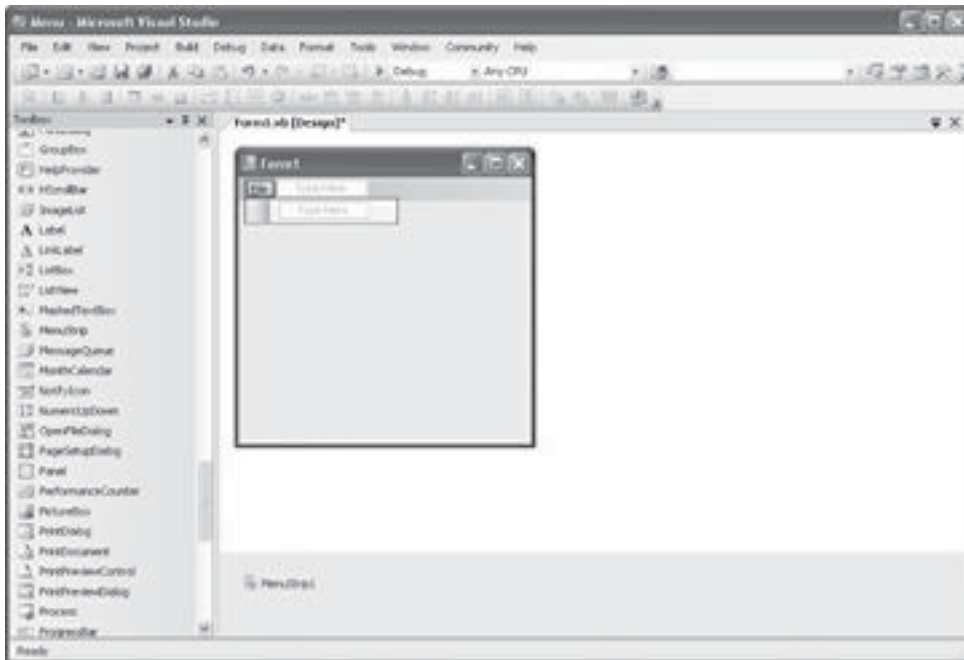


Figure 11-4 File menu item added

Tip If you forget a menu item, right-click the menu item before which the new one will be inserted and then choose *Insert | New* from the context menu. If you decide you no longer want a menu item you previously added, right-click that item and choose *Delete* from the context menu.

Items Collection Editor

One of the properties of the MenuStrip component is an Items collection, which is a collection of the ToolStripMenuItems belonging to the MenuStrip. For example, after the File and Edit menu items have been added, those menu items would belong to the Items collection of the MenuStrip.

Figure 11-5 shows the Items collection listed in the Properties window of the MenuStrip component.

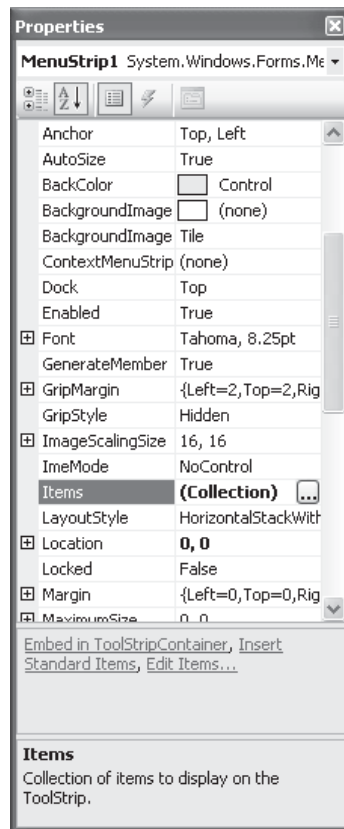


Figure 11-5 Properties window showing the Items collection of MenuStrip

Click the ellipsis (...) next to Items. This will open the Items Collection Editor, which is shown in Figure 11-6 after two `ToolStripMenuItem` (for the File and Edit menus) have been added.

You may add `ToolStripMenuItem`s to the `MenuStrip` by choosing `MenuItem` (the default selection) from the drop-down box and then clicking the Add button. Once the `ToolStripMenuItem` is added, you then may select it and in the right pane change its Name, Text, and other properties. Figure 11-6 shows properties for the Edit menu item.

You also can add menu items to the File or Edit menu item. As Figure 11-7 shows, the File menu item (as well as the Edit menu item) has a `DropDownItems` collection property. This is a collection of the `ToolStripMenuItem`s belonging to that menu item. For example, after the New and Open menu items have been added to the File menu, those menu items would belong to the `DropDownItems` collection of the File menu.

Clicking the ellipsis (...) next to `DropDownItems` will open the Items Collection Editor for that menu item. Figure 11-8 shows the Items Collection Editor for the Edit menu after menu items have been added to that menu item.

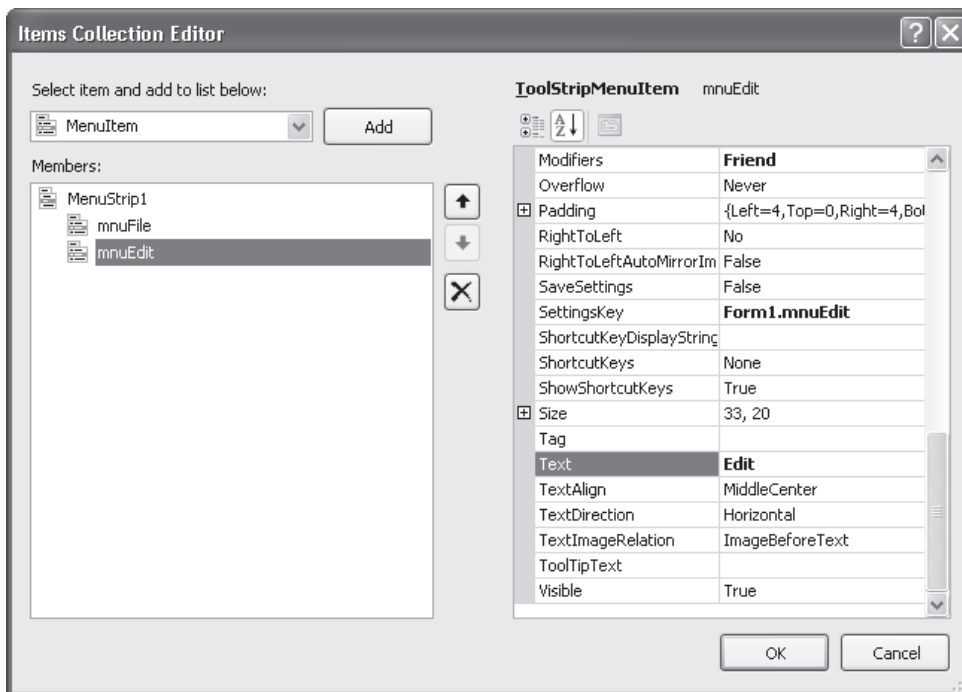


Figure 11-6 Items Collection Editor for `MenuStrip`

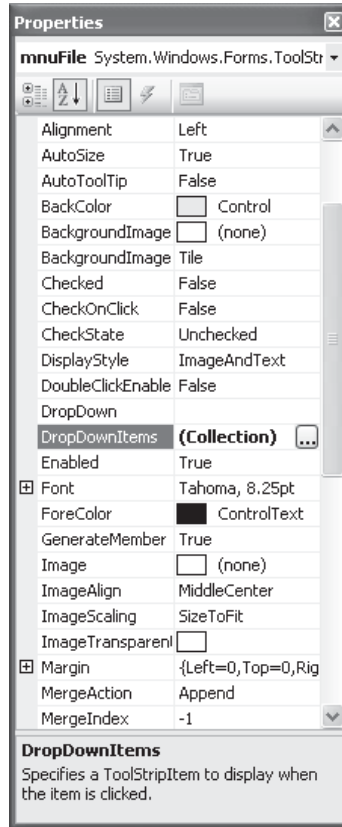


Figure 11-7 Properties window showing DropDownItems collection of the File menu item

The procedure for adding subsidiary menu items to a menu item is essentially the same as adding ToolStripMenuItems to the MenuStrip: You choose MenuItem from the drop-down box and then click the Add button. You then may select the added subitem and in the right pane change its Name, Text, and other properties.

Enhancing the Menu Items

You can enhance menu items in several ways. You can add access or shortcut keys to facilitate keyboard access to menu items. You also can add separator bars to group together related menu items.

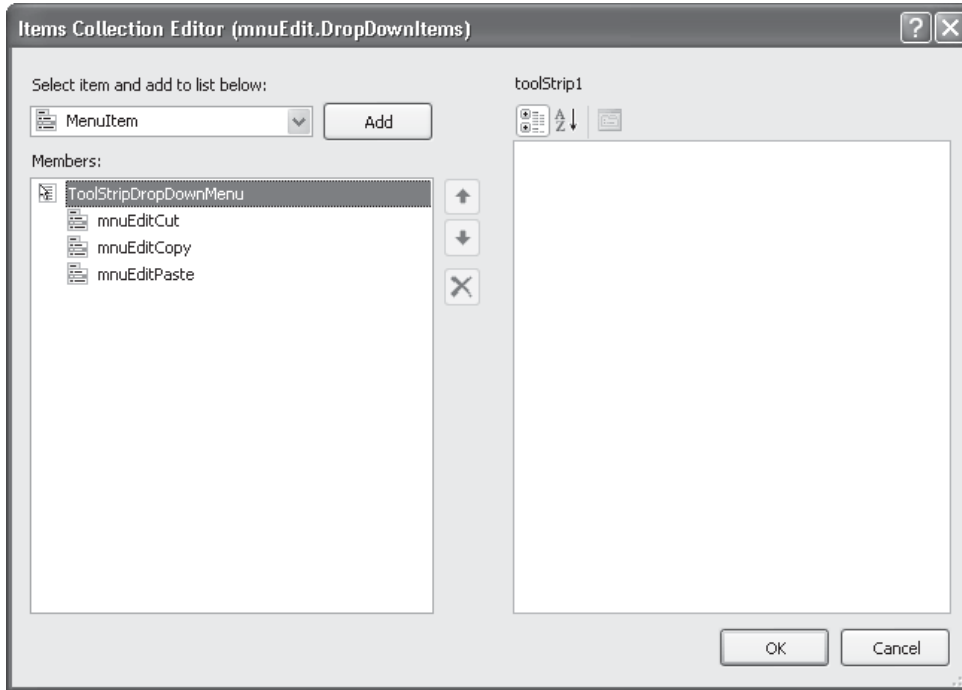


Figure 11-8 Items Collection Editor for the Edit menu item

Access Keys

Although menu items usually are accessed by a mouse click, you also should enable the user to access menu items via the keyboard. Being able to access menu items via the keyboard instead of a mouse is an important convenience, as I have discovered on an airplane flight trying to use my laptop while wedged between two sumo-sized passengers. Indeed, for users with certain disabilities, the ability to access menu items via the keyboard instead of a mouse can be a necessity.

An access key is one way of enabling the user to access menu items via the keyboard. An access key is the keyboard combination of the ALT key plus a letter in the menu item that is underlined. For example, the keyboard combination for the File menu item is ALT-F, with the *F* in File being underlined.

To add an access key, in the menu item's Text property, simply type an ampersand (&) before the letter to be underlined. Figure 11-4 earlier in this chapter shows the result of typing **&File** as the Text property for the File menu item (the *F* in File is underlined).

The access shortcut may not appear when you run the application until you press the ALT key. This is standard behavior in Windows applications. As shown in Figure 11-9, in the Effects dialog (shown by choosing the Display applet from Control Panel | Appearance tab | Effects button), the Hide Underlined Letters for Keyboard Navigation Until I Press the Alt Key option is checked by default. If you want to change that behavior, simply uncheck that box.

Shortcut Keys

Shortcut keys are another method of enabling the user to access menu items via the keyboard. In Microsoft Word, the New menu item under the File menu can be accessed with the shortcut key CTRL-N.

You can add a shortcut key at design time by selecting the menu item within the Menu Designer, selecting the ShortcutKeys property from the Properties window, and clicking the drop-down arrow. As Figure 11-10 shows, you can choose one or more of CTRL, SHIFT, or ALT by checking a box and then choosing one of the values offered in the drop-down list.

NOTE You normally would not assign a shortcut key to a top-level menu item such as File or Edit because an access key already can be used to open that menu.

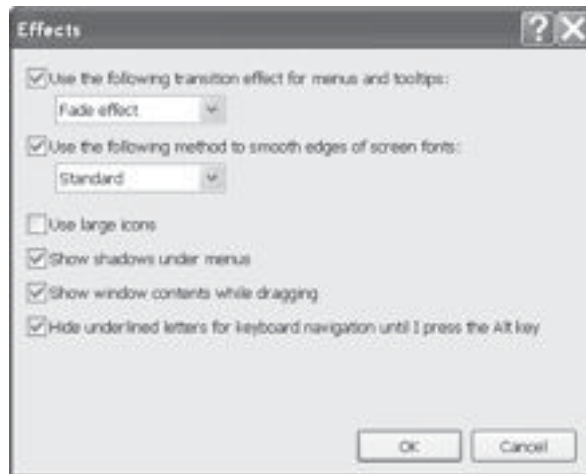


Figure 11-9 Setting whether the access shortcut is hidden until the ALT key is pressed

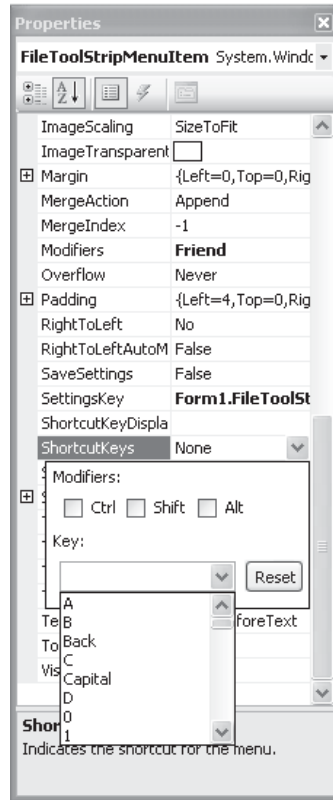


Figure 11-10 Shortcut key options displayed in the Properties window

Separator Bars

Separator bars are used to group related commands within a menu and make menus easier to read. In Microsoft Word, under the File menu, a separator bar separates the New, Open, and Close menu items from the menu items that follow them.

You may add a separator bar by setting the Text property of a menu item to a dash. Alternatively, in the Menu Designer, right-click the location where you want a separator bar and then choose Insert | Separator.

Adding Functionality to the Menu Items

The purpose of a menu item is to do something when it is clicked. Therefore, you use the Click event procedure of the menu item to provide functionality for a menu item.

The Click event, of course, occurs when the user clicks the menu item. However, the Click event also occurs if the user selects the menu item using the keyboard and presses the ENTER key, or if the user presses an access key or shortcut key that is associated with the menu item.

The Click event is not raised for all menu items. It only is raised for menu items that do not have subsidiary menu items. The reason is when a menu item with subsidiary items is clicked, the behavior is to display the subsidiary menu items. Therefore, the Click event is not raised for parent menu items such as File and Edit. Instead, the behavior when a parent menu item is clicked is to display its subitems, such as, in the case of the File menu, New, Open, and Close.

You write code for the Click event procedure for a menu item by, in code view, choosing the menu item by name from the left drop-down list and Click from the right drop-down list. You then write within the created event procedure stub the code you wish to run when the menu item is clicked. For example, the following code outputs “New” to the Output window when a menu item named mnuFileNew is clicked:

```
Private Sub mnuFileNew_Click _  
    (ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles mnuFileNew.Click  
    Debug.WriteLine("New")  
End Sub
```

Disabling Menu Items

Although menu items should be functional, there are times when you may not want them to be functional. For example, in Microsoft Word, the menu items Cut and Copy under the Edit menu initially are grayed out, or disabled. They are grayed out because no text is selected; therefore, there is nothing to cut or copy. However, once you select text, Cut and Copy are no longer grayed out—in other words, they are enabled.

A menu item should not be enabled when the command it represents is not available. It would be frustrating for the application user to click Cut or Copy and see nothing happen. The application user might be misled into thinking there is something wrong with your application. When you gray out, or disable, a menu item, the application user is given a visual cue that the menu item is not available.

Disabling a menu item that should not be available has an additional advantage—error prevention. The code for cutting text may understandably assume there is selected text. If there is no selected text, executing the code for cutting text may cause an error. By disabling the menu item when no text is selected, the code for cutting text cannot be executed when no text is selected, thus avoiding the error.

Menu items are enabled by default when they are created. However, you can disable a menu item by setting its Enabled property to False. You can do this at design time, when the menu item is selected in the Menu Designer, through the Properties window. You also can disable a menu item via code:

```
mnuFileNew.Enabled = False
```

If you want a menu item to be disabled when the application starts up, you could put this code in the Load event of the form.

Disabling the first or top-level menu item in a menu, such as the File menu item in a traditional File menu, disables all the menu items contained within the menu. Similarly, disabling a menu item that has submenu items disables the submenu items.

TIP *If all the commands on a given menu are unavailable to the user, you should hide as well as disable the entire menu. You hide the menu by setting the Visible property of the topmost menu item to False. This presents a cleaner user interface by not cluttering up your menu structure with disabled items. However, one caution: Hiding the menu alone is not sufficient to disable it. You must also disable the menu, because hiding alone does not prevent access to a menu command via a shortcut key.*

Creating a Context Menu

Many Windows applications have context menus, which are displayed when the user clicks the right mouse button over an area of the form or over a control on the form. Figure 11-11 shows a context menu in Microsoft Word.

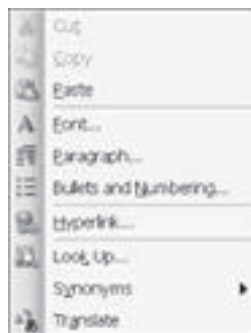


Figure 11-11 Context menu

The word “context” in context menu derives from the fact that which particular menu items are displayed often depends on the context, such as the application state, or where on the form or control the right mouse button was clicked. Indeed, in the .NET Framework, the `ContextMenuStrip` class represents shortcut or context menus.

Context menus typically are used to make available different menu items from a `MenuStrip` of a form that are useful for the user given the context of the application. For example, you can use a context menu assigned to a `TextBox` control to provide immediate access to menu items also found in the `MenuStrip` to cut, copy, and paste text, find text, change the text font, and so on.

The ability of a context menu to immediately access menu items of the main menu that might take several mouse clicks to access may be why a context menu also is called a *shortcut menu*, because the menu items on the context menu are a shortcut to menu items on the main menu. However, a context menu also may contain menu items not found in the form’s `MenuStrip`.

Adding a `ContextMenuStrip` to a Form

The process of adding a context menu to a Windows form at design time and then adding menu items to it is similar to the corresponding process discussed already in this chapter in connection with the `MenuStrip`. You first add a `ContextMenuStrip` object to your form, and then you append to it `ToolStripMenuItem` objects.

You add a context menu to a form via the following steps, which are similar to how you add a `MenuStrip` to the form:

1. View the form in designer view.
2. Double-click the `ContextMenuStrip` component in the Toolbox. As shown in Figure 11-12, this adds a `ContextMenuStrip` component to the component tray.
3. In the Properties window for that form or control, choose the `ContextMenuStrip` object (the default name may be `ContextMenuStrip1`) from the drop-down list for the form or control’s `ContextMenuStrip` property. This associates the context menu with the form or a control on the form. You also can change this value dynamically through code when the program is running if the form has more than one context menu.

Unlike with the main menu, you often will be adding a context menu to a control on the form, rather than the form itself. For example, in the Text Editor project later in this chapter, the context menu will belong to a `TextBox` control rather than the form.

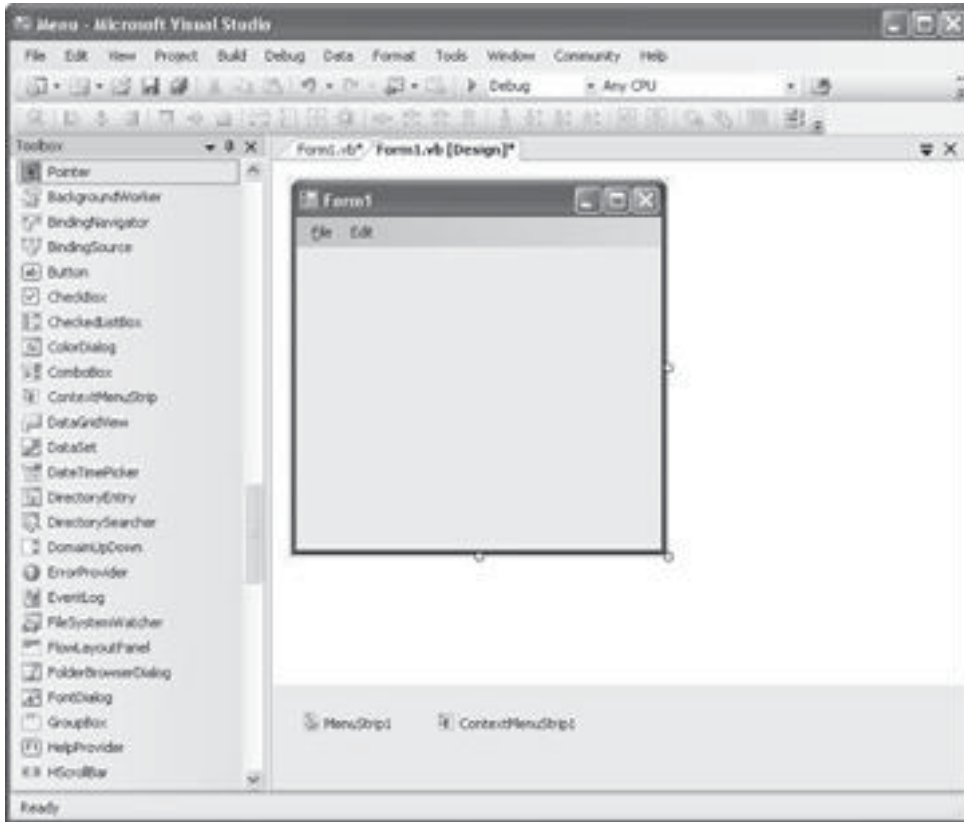


Figure 11-12 Adding a ContextMenuStrip component to a form

Adding Menu Items to the ContextMenuStrip

Once you have added a ContextMenuStrip component to your Windows form, the next step is to add menu items to it. You can do so by typing in the menu items, by using the Items Collection Editor, or by copying menu items from existing items on the main menu and pasting them onto the context menu.

Typing in the Menu Items

You can add menu items to a context menu using the same method that you used to add menu items to a main menu. You click the text “Type Here” and type the name of the desired menu item to add it. If the text “Type Here” is not displayed, you may display it by clicking the ContextMenuStrip component on the Windows form. To add another menu item, click another “Type Here” area within the Menu Designer.

You click the area below the current menu item to add another menu item, or click the area to the right of the current menu item to add submenu items.

You then should name these menu items. If the context menu item parallels one on the main menu, one naming convention is to give the context menu item the same name, other than the prefix, for which you may use “cmnu” (instead of mnu, the *c* standing for context). For example, if a context menu item parallels the main menu item Open under the File menu, named mnuFileOpen, you could name the corresponding context menu item cmnuFileOpen.

NOTE One difference between a context menu and a main menu is that a context menu usually does not have a top-level item, such as File in the main menu.

Items Collection Editor

You also can use the Items Collection Editor to add items to a context menu as well as to the main menu.

Figure 11-13 shows the Properties window for the ContextMenuStrip.

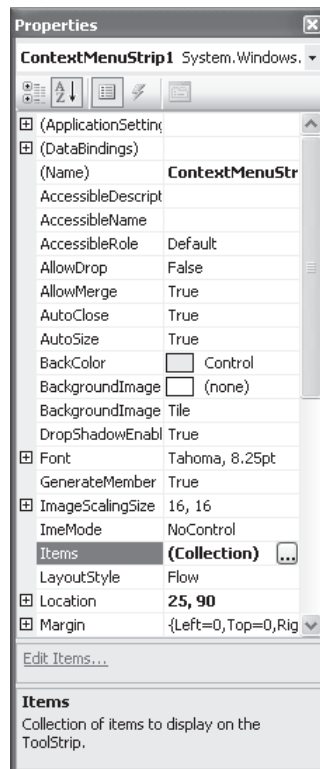


Figure 11-13 Properties window for the ContextMenuStrip

You also can add items to the `ContextMenuStrip`. As Figure 11-13 shows, the `ContextMenuStrip` has an `Items` collection property.

Clicking the ellipsis (...) next to `Items` will open the `Items Collection Editor` for the `ContextMenuStrip`, which is shown in Figure 11-14 after menu items have been added to the `ContextMenuStrip`.

You add `ToolStripMenuItem`s to the `ContextMenuStrip` by choosing `MenuItem` from the drop-down box and then clicking the `Add` button. You then may select the added `ToolStripMenuItem` and in the right pane change its `Name`, `Text`, and other properties. Figure 11-14 shows the properties for the first menu item on the context menu.

Copying and Pasting

You may want the context menu to duplicate commands in the main menu. For example, the `Cut`, `Copy`, and `Paste` menu commands in Microsoft Word's `Edit` menu are often duplicated in a menu when you click on the document.

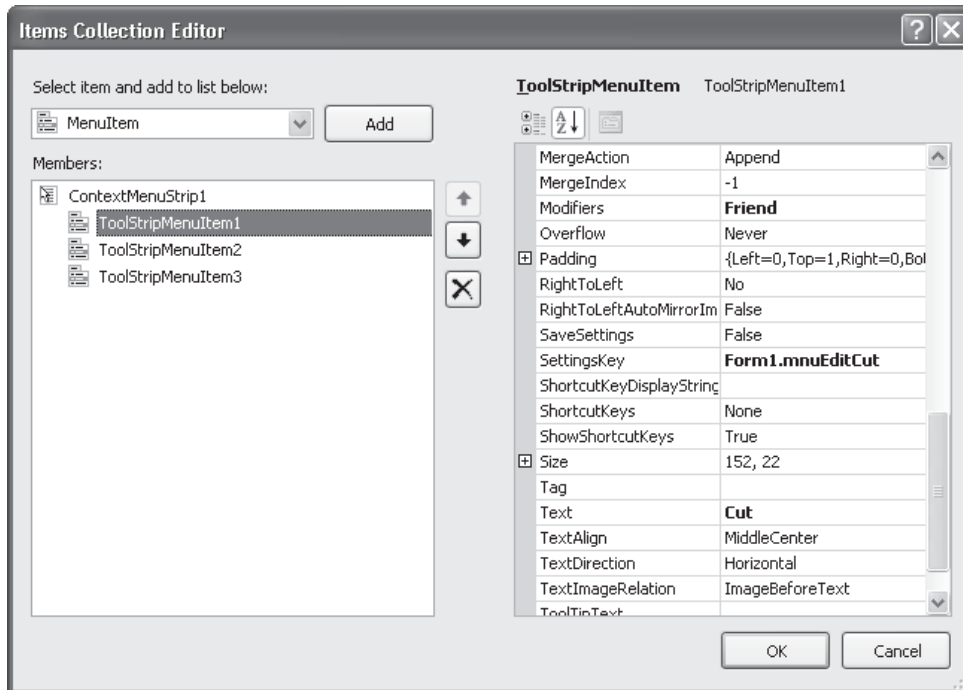


Figure 11-14 Items Collection Editor for `ContextMenuStrip`

You do not need to re-create the entire menu structure when you want to duplicate a given menu's functionality. You may use the Menu Designer to copy menus by following these steps:

1. Within the Menu Designer, choose the MenuStrip component, select the menu item or items (using the **SHIFT** key for multiple items) you would like to duplicate, right-click them, and choose **Copy**, as shown in Figure 11-15.
2. Choose the ContextMenuStrip component, select the "Type Here" area where you would like the first menu item to appear, and then right-click and choose **Paste**, as shown in Figure 11-16.
3. Figure 11-17 shows the end result.

Adding Functionality to Context Menu Items

You add functionality to menu items in a ContextMenuStrip the same way as you add functionality to menu items in a MenuStrip—by using the Click event procedure of the menu item.

Often a context menu item corresponds to a menu item on the main menu. For example, on the main menu, you may have an **Edit | Select All** menu item, and on a context menu, you may have a **Select All** context menu choice. If the user chooses

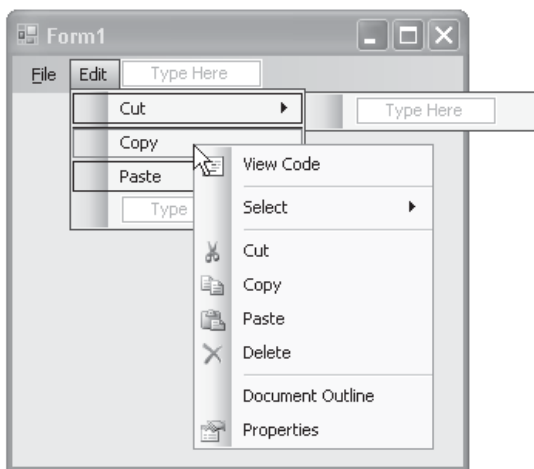


Figure 11-15 Copying items from the MenuStrip

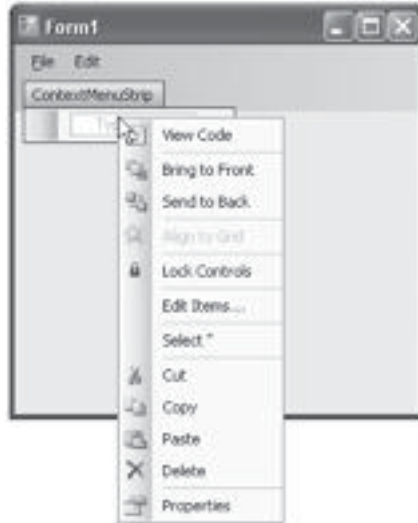


Figure 11-16 Pasting items into the ContextMenuStrip

Select All from the context menu, rather than writing a duplicate event procedure, you want the Click event procedure of the Edit | Select All menu item to run. You have three alternatives for having the Click event procedure for the main menu item also handle the Click event for the corresponding context menu item.

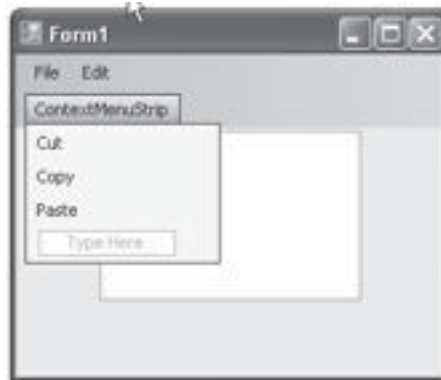


Figure 11-17 Context menu now populated

AddHandler

The first alternative is to use the `AddHandler` keyword. This keyword associates an event of a control to an event procedure that follows yet another keyword, `AddressOf`. The following code assumes the context menu item is named `cmnuEditSelectAll` and the corresponding main menu item is named `mnuEditSelectAll`:

```
AddHandler cmnuEditSelectAll.Click, _  
    AddressOf Me.mnuEditSelectAll_Click
```

This code designates the `Click` event procedure of `mnuEditSelectAll` as handling the `Click` event of `cmnuEditSelectAll`. This code logically could be placed in the `Load` event of the form.

`AddHandler` is used when you may not know at design time the precise event handler that will be used at run time because, for example, the choice will depend on user actions. One of the following two alternatives should be used instead if the event handler is known at design time because `AddHandler` has a greater performance cost.

Handles Clause

The second alternative is to expand the `Handles` clause of the `Click` event procedure of the main menu item (here, `mnuEditSelectAll`). This event procedure already has the clause `Handles mnuEditSelectAll.Click`. You add `cmnuEditSelectAll.Click` to the `Handles` clause, using a comma to separate it from `mnuEditSelectAll.Click`, as shown here:

```
Private Sub mnuEditSelectAll_Click _  
    (ByVal sender As Object, _  
    ByVal e As System.EventArgs) _  
    Handles mnuEditSelectAll.Click, cmnuEditSelectAll.Click  
    txtEdit.SelectAll()  
End Sub
```

Calling Another Event Procedure

The third alternative is to call the `Click` event procedure of the main menu item from the `Click` event procedure of the context menu item:

```
Private Sub cmnuEditSelectAll_Click _  
    (ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) _  
    Handles cmnuEditSelectAll.Click  
    mnuEditSelectAll_Click(sender, e)  
End Sub
```

NOTE You must pass the arguments *sender* and *e* to the `mnuEditSelectAll_Click` call because the `Click` event procedure of that main menu item expects those arguments.

Text Editor Project

This project is a text editor. The application user can type and use the main menu or the context menu to cut, copy, and paste. Figure 11-18 shows the Text Editor project at run time with the context menu displayed.

Creating the Project

You can create the Text Editor project with the following steps:

1. Create a new Windows application.
2. Add a `TextBox` control to the form from the Toolbox. Name it `txtEdit`, set its `Multiline` property to `True`, and delete any text in its `Text` property. You also should resize the control so it is large enough to show multiple lines of text.
3. Add a `MenuStrip` component to the form from the Toolbox.
4. Using the Menu Designer, add a menu whose top-level menu item is `Edit` and subsidiary menu items are `Cut`, `Copy`, and `Paste`. Name the `Edit` menu item `mnuEdit`, the `Cut` menu item `mnuEditCut`, the `Copy` menu item `mnuEditCopy`, and the `Paste` menu item `mnuEditPaste`.
5. Using the Properties window, set the `MainMenuStrip` property of the form to the name of your `MenuStrip` component.

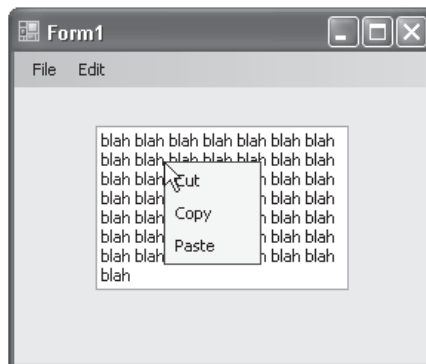


Figure 11-18 Text Editor project at run time

6. Add a ContextMenuStrip component to the form from the Toolbox.
7. Using the Properties window, set the ContextMenuStrip property of the text box to the name of your ContextMenuStrip component. Also set the ShowImageMargin property of the ContextMenuStrip control to False so the context menu will not have a left-hand margin.
8. Copy the Cut, Copy, and Paste menu items from the MenuStrip to the ContextMenuStrip. Name these menu items in the context menu cmnuEditCut, cmnuEditCopy, and cmnuEditPaste, respectively.
9. In the Code editor, create a Click event procedure for the Edit | Cut menu item (mnuEditCut) and write the following code in it:

```
Private Sub mnuEditCut_Click (ByVal sender As Object, _  
    ByVal e As System.EventArgs) Handles mnuEditCut.Click  
    txtEdit.Cut()  
End Sub
```

10. In the Code editor, create a Load event procedure for the form and write the following code in it:

```
AddHandler cmnuEditCut.Click, _  
    AddressOf Me.mnuEditCut_Click
```

11. In the Code editor, create a Click event procedure for the Edit | Copy menu item (mnuEditCopy) and write the following code in it:

```
Private Sub mnuEditCopy_Click (ByVal sender As Object, _  
    ByVal e As System.EventArgs) _  
    Handles mnuEditCopy.Click, cmnuEditCopy.Click  
    txtEdit.Copy()  
End Sub
```

12. In the Code editor, create a Click event procedure for the Edit | Paste menu item (mnuEditPaste) and write the following code in it:

```
Private Sub mnuEditPaste_Click (ByVal sender As Object, _  
    ByVal e As System.EventArgs) Handles mnuEditPaste.Click  
    txtEdit.Paste()  
End Sub
```

13. In the Code editor, create a Click event procedure for the Edit | Paste context menu item (cmnuEditPaste) and write the following code in it:

```
Private Sub cmnuEditPaste_Click (ByVal sender As Object, _  
    ByVal e As System.EventArgs) Handles cmnuEditPaste.Click  
    mnuEditPaste_Click(sender, e)  
End Sub
```

Explanation of the Code

The `TextBox` class has `Cut`, `Copy`, and `Paste` methods. These methods work the same as the `Cut`, `Copy`, and `Paste` menu items of the `Edit` menu in Microsoft Word and other Windows applications. The `Cut` method copies the selected text to the clipboard, but removes the selected text from the text box. The `Copy` method also copies the selected text to the clipboard, but does not remove the selected text from the text box. The `Paste` method copies the text in the clipboard to the text box, beginning with the cursor location in the text box.

The `Cut`, `Copy`, and `Paste` methods of the `TextBox` class are called in the `Click` event procedures of the corresponding `Edit` menu items: `Edit | Cut` (`mnuEditCut`), `Edit | Copy` (`mnuEditCopy`), and `Edit | Paste` (`mnuEditPaste`).

The `Cut`, `Copy`, and `Paste` methods of the `TextBox` class also could be called in the `Click` event procedures of the corresponding context menu items: `Cut` (`cmnuEditCut`), `Copy` (`cmnuEditCopy`), and `Paste` (`cmnuEditPaste`). However, this would be a duplication of code. Here, the duplication is short, but in other circumstances it may not be. Therefore, it is useful instead to have each context menu item's functionality handled by the corresponding `Edit` main menu item.

The earlier section “Adding Functionality to `ContextMenuStrip` Menu Items” discussed three different alternatives for having a context menu item's functionality handled by the corresponding main menu item. To illustrate the use of all three alternatives, the `AddHandler` alternative is used for the `Cut` context menu item, the `Handles` alternative is used for the `Copy` context menu item, and the alternative of calling another event procedure is used for the `Paste` context menu item.

NOTE *AddHandler is used here just for illustration. As mentioned already, because of its greater performance cost, you should use it only when you may not know at design time the precise event handler that will be used at run time (for example, because the choice will depend on user actions). Here, the correct event handler is known at design time.*

Run the application. Type some text in the text editor, select some text, and then cut, copy, and paste using the main menu and the context menu.

This text editor certainly is not ready for the commercial market. The `Cut`, `Copy`, and `Paste` items need to be disabled at the appropriate times. Additionally, further commands are needed, such as `Undo`, `Select All`, and so on. Nevertheless, the `Text Editor` project is useful in demonstrating how to link corresponding items on a main menu and a context menu, as well as showing some methods of the `TextBox` control.

Conclusion

Application users need to give commands to the application, such as to open, save, or close a file, print a document, cut, copy, or paste text, and so on. Application users give such commands through the GUI of the application. Two of the most common GUI elements through which application users give commands to an application are the main menu and the context or shortcut menu. In this chapter, you learned how to create them and to handle and link their events.

There is another common GUI element through which application users also give commands to an application—toolbars. In the next chapter, you will learn how to create toolbars and coordinate them with your menus.

Quiz

1. What class represents a main menu?
2. What class represents each item on a main menu?
3. What is an access key?
4. Is the Click event raised for all menu items?
5. How do you gray out a menu item so it is not available when it should not be?
6. What does the Items collection of the MenuStrip component contain?
7. What class represents the shortcut or context menu?
8. What class represents each item on a context menu?
9. What does the Items collection of the ContextMenuStrip component contain?
10. What are different alternatives for having a context menu item's functionality handled by the corresponding main menu item?

Toolbars

This chapter is all about bars, but not the kind that inspired the song “Looking for Love in All the Wrong Places.” In this chapter, we’ll explore a kind of bar that will enable you to enhance your application both visually and functionally.

The toolbar is a part of every Windows programmer’s life. You would be hard-pressed to find a Windows application that doesn’t have a toolbar. Indeed, most Windows applications have several of them.

The functionality of a toolbar button generally duplicates the functionality of a menu item. For example, the toolbar button with the printer icon duplicates the functionality of the File | Print menu item.

There are two good reasons for using a toolbar even though it may duplicate the functionality of a menu. First, the buttons on the toolbar are immediately accessible. By contrast, the items on the menus may be nested several levels deep and can be accessed only by multiple mouse clicks or keystrokes. Second, a toolbar button usually has an image, whereas a menu item usually is text. Quite simply, visual items are more attractive and apparent to the application’s user than text items. This is *Visual Basic*, after all!

This chapter will show you, through enhancing the Text Editor project you created in Chapter 11, how to create toolbars for your forms, add buttons to them, and add images to the buttons. You also will learn how to associate the clicking of a particular toolbar button with the clicking of a corresponding menu item.

Creating a Toolbar

Just as the main menu is represented by the `MenuStrip` class, the toolbar is represented by the `ToolStrip` class. A `ToolStrip` object contains a collection of buttons or other types of controls.

Creating a toolbar is a two-step process: First, you add a `ToolStrip` object to your form. Second, you add buttons or other controls to the toolbar.

Adding a Toolbar to a Form

You add a `ToolStrip` object to a form using the following steps, similar to adding a `MenuStrip` object to a form. Try the following steps to add a `ToolStrip` to the Text Editor project that you created in Chapter 11:

1. Open the Text Editor project.
2. Open the form in designer view.
3. Double-click the `ToolStrip` component in the Toolbox to add it to the form. Figure 12-1 shows the `ToolStrip` component after it has been added to the form.

As Figure 12-1 shows, the `ToolStrip` control, like the `MenuStrip` and `ContextMenuStrip` components, appears in the component tray. The `ToolStrip` control also appears as a large gray area under the menu area. This is where the toolbar will be located.

Figure 12-2 shows that, when the `ToolStrip` control has focus or you click the four vertical dots on the left side of the `ToolStrip` control, a drop-down box appears on the left side of the `ToolStrip` control, and what is called a *smart task arrow* appears on the right side of the `ToolStrip` control.

The `ToolStrip` control is automatically associated with the form. This is unlike the `ContextMenuStrip` component, which is not associated with the form without you first setting the `ContextMenuStrip` property of the form.

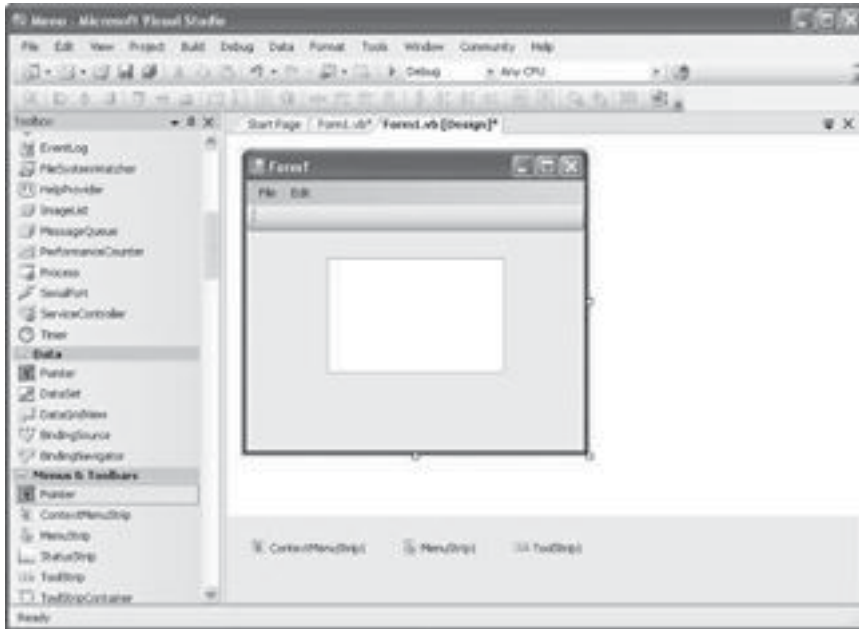


Figure 12-1 ToolStrip component added to the form

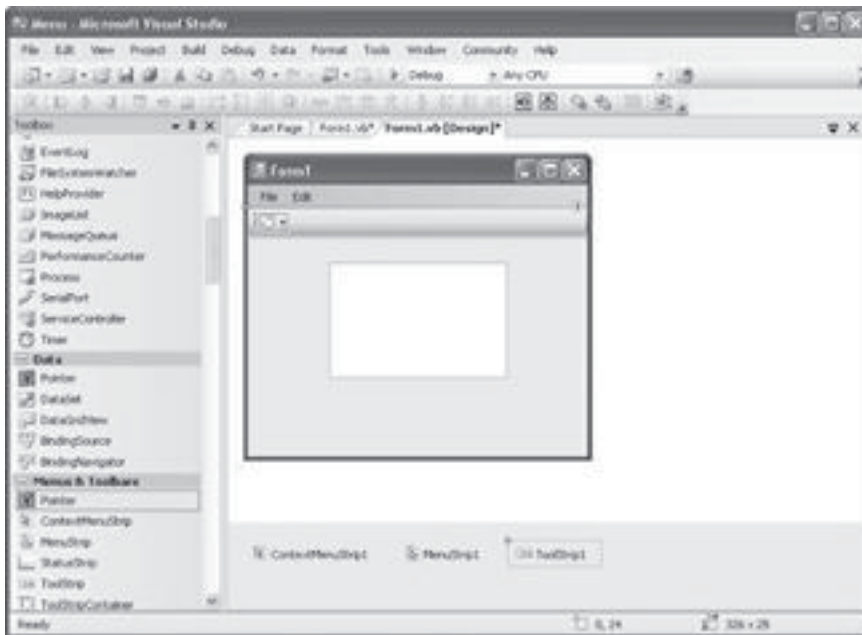


Figure 12-2 ToolStrip with drop-down box and smart task arrow

NOTE The toolbar we just added has the default name of `ToolStrip1`. You don't need to change this name because this project uses only one toolbar. However, if your application uses more than one toolbar, as many applications do, then you should choose logical names to differentiate among the different toolbars.

Adding Buttons to the Toolbar

The Button control, represented by the `ToolStripButton` class, is the most common type of control on a toolbar, and therefore it's the control covered in this section. However, toolbars may contain other types of controls. For example, in Microsoft Word, the formatting toolbar contains drop-down boxes for the type and size of fonts.

There are several different alternative methods by which you can add buttons and other controls to the toolbar. One alternative is the Items Collection Editor, which we used in Chapter 11 to add items to the main menu. Figure 12-3 shows the Items Collection Editor for the toolbar.

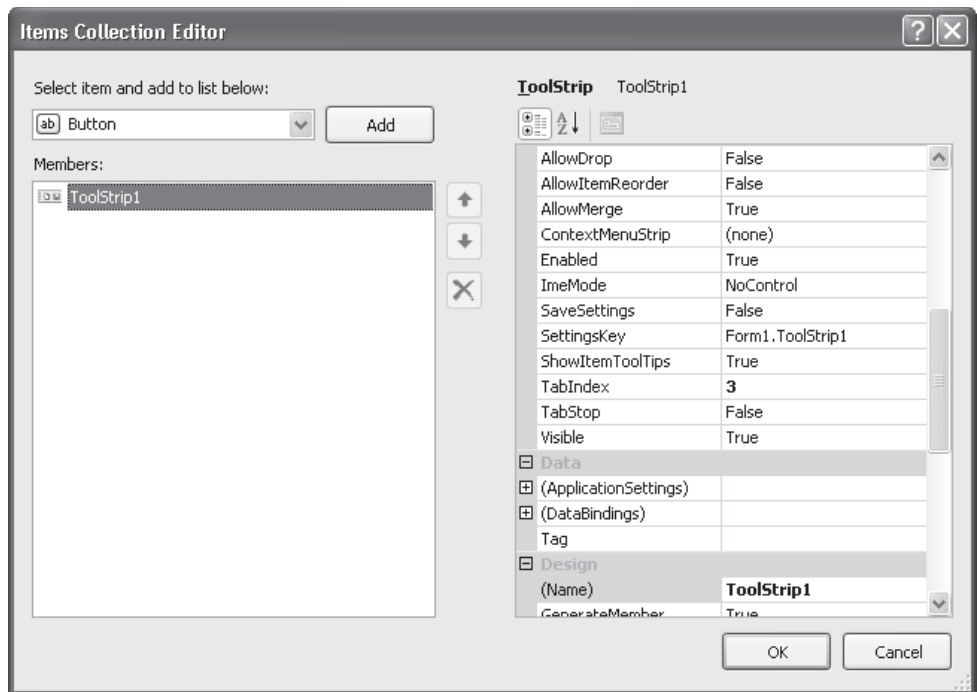
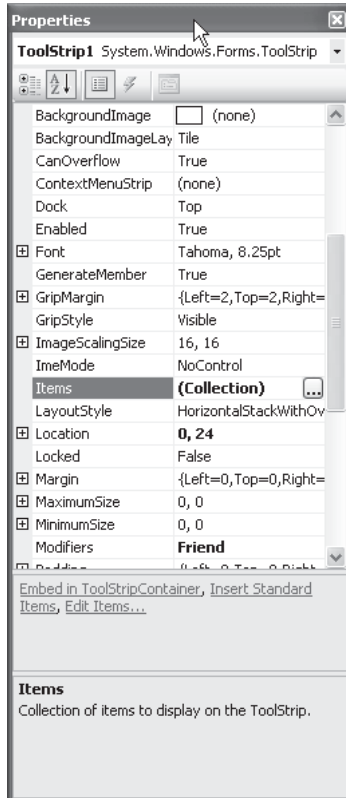
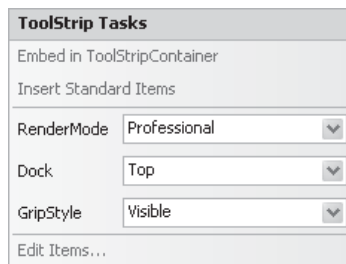


Figure 12-3 Items Collection Editor for the toolbar

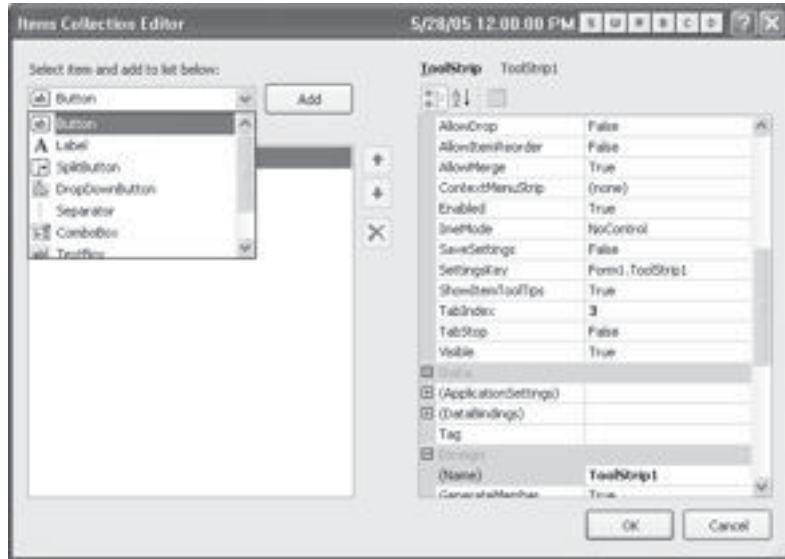
You can display the Items Collection Editor by displaying the Properties window for the toolbar and then clicking the ellipsis (...) next to the Items collection property shown here:



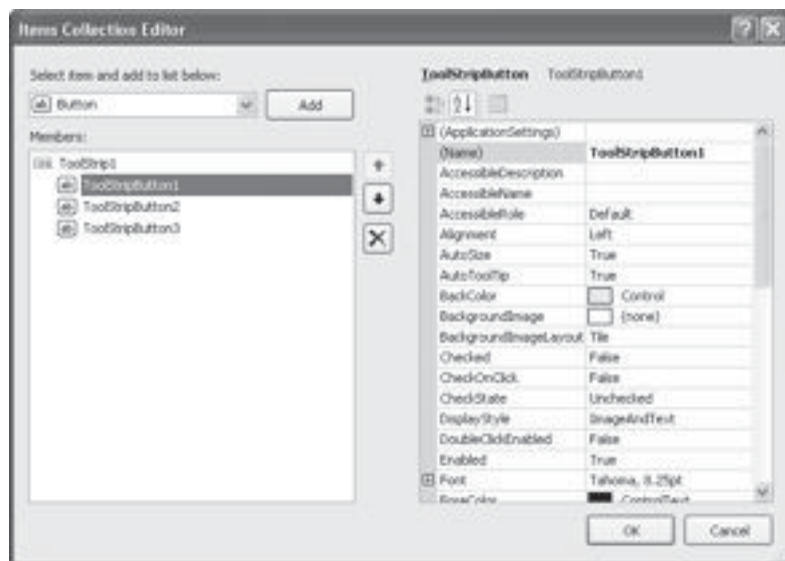
You also can display the Items Collection Editor by first clicking the smart task arrow at the rightmost edge of the toolbar. This displays the ToolStrip Tasks pane shown next. Clicking Edit Items... at the bottom of this pane displays the Items Collection Editor.



Once you display the Items Collection Editor, you first select the type of item to be added. The item usually is a Button control, but also may be another type of control, as shown here.



Once you have chosen the control, you then click the Add button to add the control to the toolbar. The following screenshot shows the Items Collection Editor after three buttons have been added to the toolbar.



As illustrated here, choosing one of the buttons in the left pane shows the button's properties in the right pane. You should change each button's Name property. Later in this chapter, I will be using these buttons to parallel the functionality of the Edit | Cut, Edit | Copy, and Edit | Paste menu items. Accordingly, I named the three buttons `tbtnEditCut`, `tbtnEditCopy`, and `tbtnEditPaste`. The "tbtn" prefix indicates a toolbar button, and `EditCut` (or `EditCopy` or `EditPaste`) indicates the functionality of the toolbar button.

Additionally, you should delete the value of the Text property of each button because these buttons will be displaying images, not text.

Click OK to close the Items Collection Editor and create the buttons you specified. Figure 12-4 shows the toolbar area after several buttons have been added.

Associating Images with Toolbar Buttons

So far our toolbar is not very impressive. All the buttons look the same, with a generic image that, as near as I can tell, looks like a sun over a mountain.

The most common visual cue for a toolbar button is an image. Figure 12-5 shows a toolbar in Microsoft Word. The images show each toolbar button's purpose, such as New, Open, and Save.

We are now going to add images to the toolbar buttons for this project.

The first step concerns the `DisplayStyle` property of the `ToolStripItem` class. This property, which is an enumeration, determines whether an image or text may be displayed on a button. Table 12-1 lists the possible values for this property.



Figure 12-4 Toolbar with added buttons



Figure 12-5 Images on toolbar buttons in Microsoft Word

Using the Items Collection Editor, set each button’s DisplayStyle property to Image (if necessary, since it is the default) because we intend each button to display an image but no text. Text is helpful to identify the purpose of a toolbar button. However, the small area of the button would be crowded by including text as well as an image.

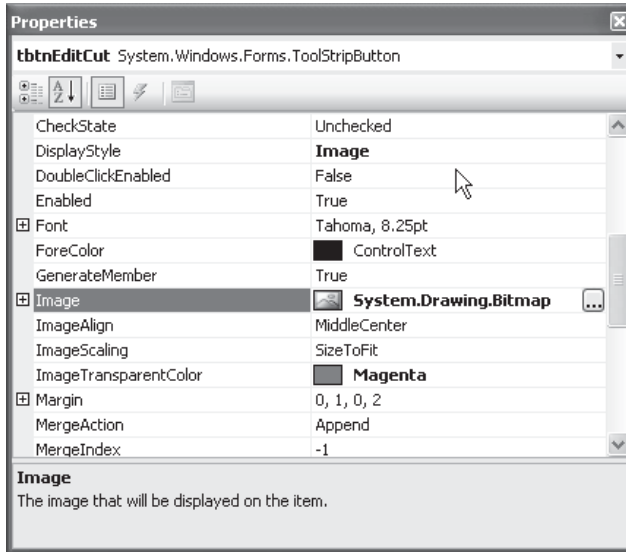
NOTE You can set the *ToolTipText* property of the button to a short textual hint of the button’s purpose. For example, you could set the *ToolTipText* property of *tbnEditCut* to “Cut.” Then, when the user hovers the mouse cursor over the button, a *ToolTip* of “Cut” will appear. A *ToolTip* has the advantage of providing a textual explanation of the button’s purpose without taking up space on the small area of the button.

The next step is to set the Image property of each button. This property, as its name suggests, sets the image to be displayed in the button.

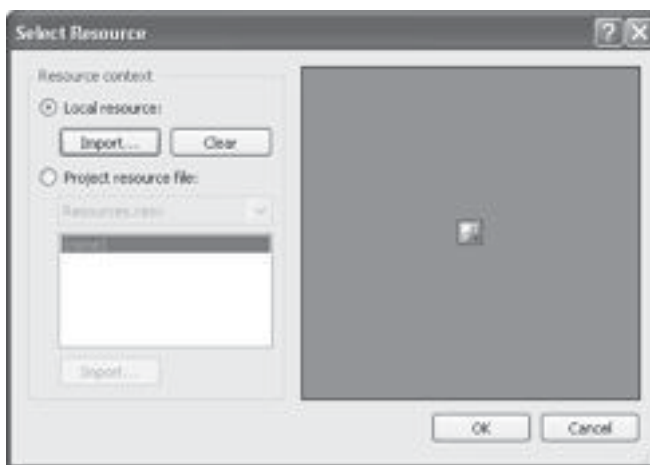
Value	Description
Image	The ToolStripItem may display only an image, which is the default.
ImageAndText	The ToolStripItem may display both an image and text.
None	The ToolStripItem may not display either an image or text.
Text	The ToolStripItem may display only text.

Table 12-1 DisplayStyle Enumeration Values

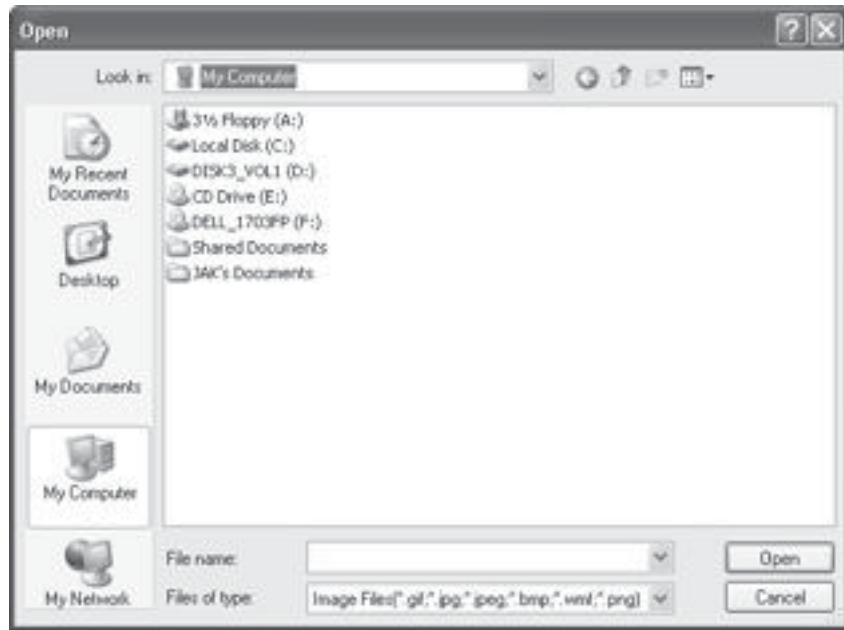
Using the Items Collection Editor, go to the Image property of a button. Here, you can see the Image property of the Cut button, which currently is set to `System.Drawing.Bitmap` and shows the default image.



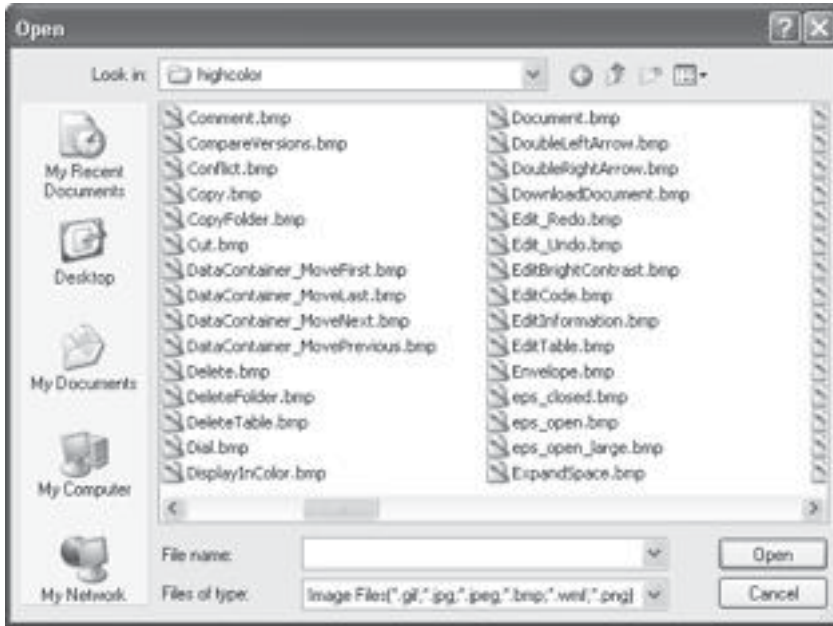
Click the ellipsis (...) next to `System.Drawing.Bitmap`. This will display the Select Resource dialog box shown here. You use this dialog box to assign an image to a form or control in a Windows application.



Choose the Local Resource radio button and then click the Import button associated with it. This displays the Open dialog box shown next, which you use to browse to and select an image file to be displayed on the button.

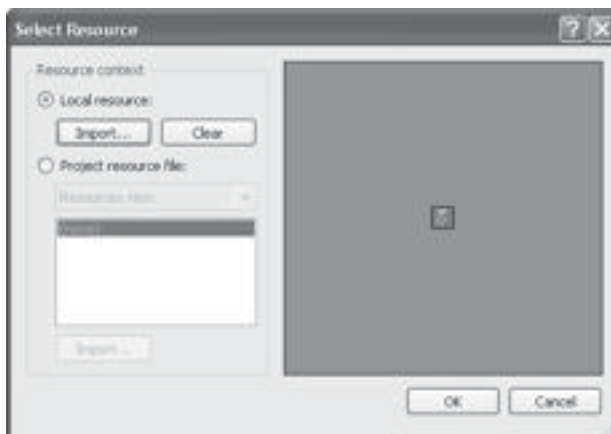


Visual Studio 2005 includes bitmap files you can use as toolbar images. These files are located by default within the directory `C:\Program Files\Microsoft Visual Studio 8\Common7\VS2005ImageLibrary`. From there I went to the folder `bitmaps\commands\highcolor`, shown here. As you can see, there are bitmap files (.bmp extensions) for Cut, Copy, and (if you scroll further in the dialog box depicted in this screenshot) Paste.

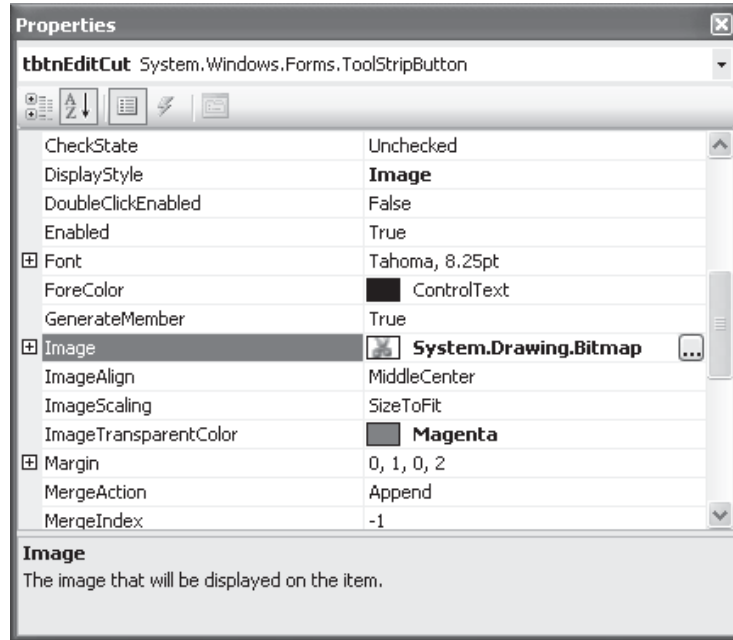


NOTE You may not have these bitmap files installed, or they may be installed at a different location, depending on the particular edition you purchased or your installation options.

Choose the Cut bitmap file for the Cut toolbar button and then click the Open button. As shown here, the Select Resource dialog box now contains the image for Cut.



Click OK in the Select Resource dialog box. As shown here, the Items Collection Editor now shows an image for the Image property of the Cut button.



Repeat the same process for the Copy and Paste buttons, except of course choose Copy.bmp for the Copy button and Paste.bmp for the Paste button. When you're done, click OK to close the Items Collection Editor. Figure 12-6 shows the toolbar, with images for Cut, Copy, and Paste.



Figure 12-6 Toolbar buttons with images for Cut, Copy, and Paste

NOTE *The size of the bitmap and the size of the toolbar button may be different. You can set the `ImageScaling` property to `SizeToFit` so the image will size to fit on the toolbar button.*

Associating Code with Clicks of Toolbar Buttons

The toolbar buttons look prettier now that each has an image on it, but they still don't do anything when they're clicked.

In this section, we'll write code so the Cut toolbar button provides the same cut action as the Cut menu item and context menu item we worked on in Chapter 11. Similarly, when you're finished with this section, the Copy toolbar button will provide the same copy action as the Copy menu item and context menu item, and the Paste toolbar button will provide the same paste action as the Paste menu item and context menu item.

The Cut, Copy, and Paste methods of the `TextBox` class also could be called in the Click event procedures of the corresponding toolbar buttons. However, as discussed in Chapter 11 in connection with context menu items, this would be a duplication of code. Here, the duplication is short, but in other circumstances it may not be. Therefore, it is useful instead to have each toolbar button's functionality handled by the corresponding Edit main menu item.

Chapter 11, in the section "Adding Functionality to Context Menu Items," discussed three different alternatives for having a context menu item's functionality handled by the corresponding main menu item. The same discussion applies here to having a toolbar button's functionality handled by the corresponding main menu item. To illustrate the use of all three alternatives, the `AddHandler` alternative is used for the Cut toolbar button, the `Handles` alternative is used for the Copy toolbar button, and the alternative of calling another event procedure is used for the Paste toolbar button.

Add the following line of code to the Load event procedure for the form so the Click event procedure of the Edit | Cut menu item handles the Click event of the Cut toolbar button:

```
AddHandler tbtnEditCut.Click, _  
    AddressOf Me.mnuEditCut_Click
```

Expand the `Handles` clause of the Click event procedure for the Edit | Copy menu item (`mnuEditCopy`) so it also handles the Click event of the Copy toolbar button:

```
Private Sub mnuEditCopy_Click(ByVal sender As Object, _  
    ByVal e As System.EventArgs) _
```

```
Handles mnuEditCopy.Click, _  
    cmnuEditCopy.Click, tbtnEditCopy.Click  
    txtEdit.Copy()  
End Sub
```

Finally, create a Click event procedure for the Paste toolbar button so it calls the Click event procedure of the Edit | Paste menu item (mnuEditPaste):

```
Private Sub tbtnEditPaste_Click(ByVal sender As Object, _  
    ByVal e As System.EventArgs) Handles tbtnEditPaste.Click  
    mnuEditPaste_Click(sender, e)  
End Sub
```

Conclusion

Application users use toolbars as well as menu items to give commands to an application. The functionality of a toolbar button generally duplicates the functionality of a menu item. However, the purpose of this duplication is that toolbar buttons have two advantages over menu items. First, toolbar buttons are immediately accessible, whereas menu items may be nested several levels deep and can be accessed only by multiple mouse clicks or keystrokes. Second, a toolbar button is visual, which gives a more visual interface than the text of a menu item.

This chapter showed you how to create toolbars for your forms, add buttons to them, and add images to the buttons. Transitioning from the graphical user interface to code, you also learned how to associate the clicking of a particular toolbar button with the clicking of a corresponding menu item.

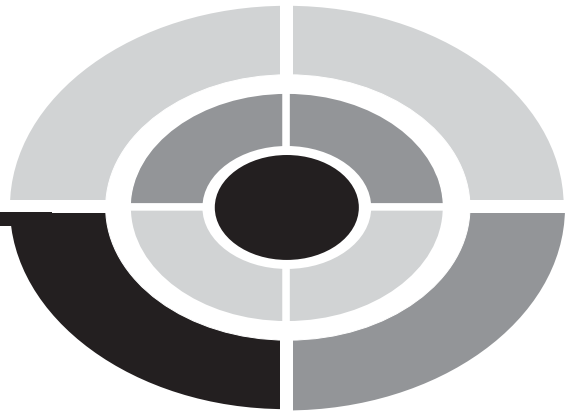
So far our text editor is not able to read from or write to any file from the hard drive. This functionality will be added in the next chapter.

Quiz

1. What class represents a toolbar?
2. What class represents each item on a toolbar?
3. What does the Items collection of the ToolStrip component contain?
4. Is a toolbar item limited to a button?
5. What are advantages of a toolbar over a corresponding menu?

6. What are different alternatives for having a toolbar item's functionality handled by the corresponding main or context menu item?
7. What does the `DisplayStyle` property of the `ToolStripItem` class determine?
8. What does the `Image` property of the `ToolStripItem` class determine?
9. What editor is useful in adding controls to a toolbar?
10. What is a good prefix for naming a toolbar button?

This page intentionally left blank



PART FIVE

Accessing Data

This page intentionally left blank

Accessing Text Files

Perhaps the most common purpose of Visual Basic applications is to access, view, and modify data. The data is stored on the computer's hard drive as a file or files so the data will be available even after the application exits.

Text files long have been used to store data. Text files preceded databases, but they often are not thought of as advanced as databases such as Oracle, SQL Server, and Access. Indeed, databases do have advantages over text files. However, unlike databases, which each has a different format and therefore often can be understood only by applications that have the software for that particular database format, text files generally are universally understood by applications. For this reason, text files are used as a common language between applications that otherwise have incompatible software for data transfer between them.

I will show you in this chapter how to read from and write to a text file. First, however, I will show you how to add to your program Open and Save dialog boxes, such as those used in sophisticated programs like Microsoft Word, so you can open a text file to read from it as well as save to a text file to write to it.

Open and Save File Dialog Boxes

In Microsoft Word and many other Windows programs, the application user may open a file located with the Open dialog box, which they display with the File | Open menu command or the Open toolbar button. Similarly, the application user may save information to a file with the Save dialog box, which they display with the File | Save menu command or the Save toolbar button.

The Open dialog box is a control of the OpenFileDialog class, and the Save dialog box is a control of the SaveFileDialog class. In this section, I will show you how to add Open and Save dialog boxes to your application.

Adding an OpenFileDialog Control to Your Form

Figure 13-1 shows an Open dialog box in Notepad.

You add an OpenFileDialog control to a form using the following steps, similar to adding a MenuStrip or ToolStrip object to a form. Try the following steps to add an OpenFileDialog control to the Text Editor project that you created in Chapter 11 and enhanced in Chapter 12:

1. Open the Text Editor project.
2. Open the form in designer view.
3. Double-click the OpenFileDialog control in the Toolbox (it is in the Dialogs section) to add it to the form.

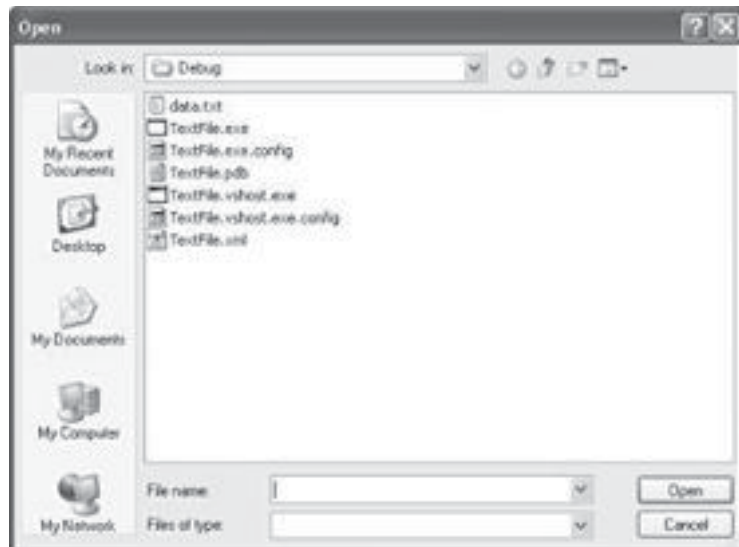


Figure 13-1 Open dialog box in Notepad

Figure 13-2 shows the OpenFileDialog control after it has been added to the form. OpenFileDialog won't appear directly on your form, but instead in the component tray below the form, as shown in Figure 13-2.

The default name of this control likely is OpenFileDialog1. Give this control a more logical name, such as dlgOpen. The “dlg” prefix indicates the control is a dialog box, and Open indicates that the purpose of the dialog box is to open a file. You should also change the FileName property so that it doesn't display the control's name in the dialog box. You don't need to change any of the other default properties of this control.

Showing the OpenFileDialog Control

The MenuStrip, ContextMenuStrip, and ToolStrip controls also appear in the component tray. However, unlike these controls, the OpenFileDialog control won't appear on your form when you run your program. Instead, you need to write code to display the OpenFileDialog control.

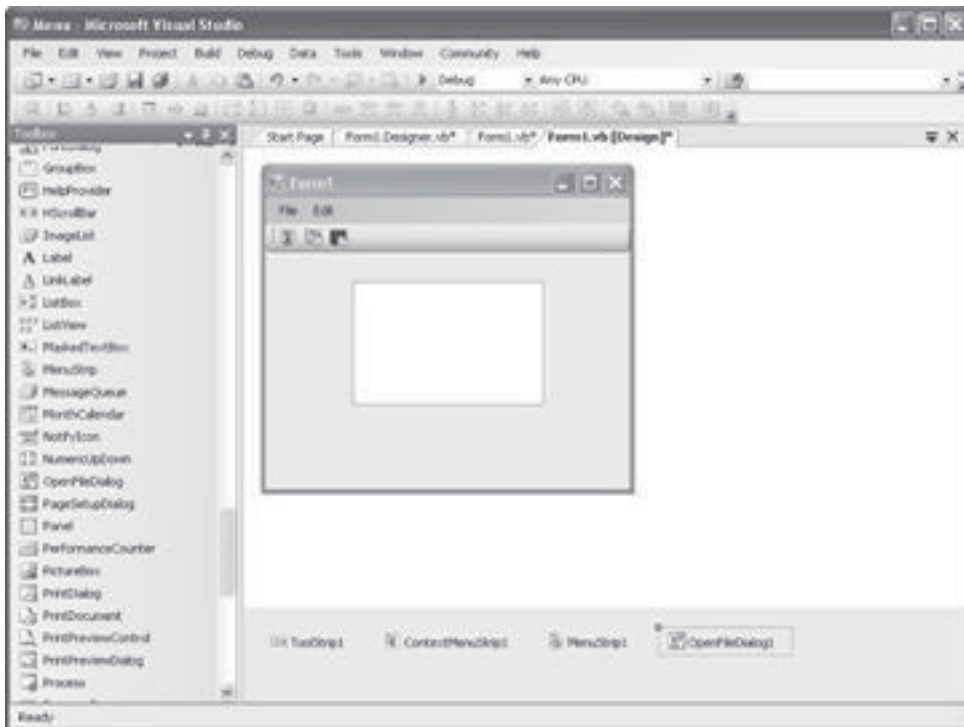


Figure 13-2 OpenFileDialog in the component tray

One of the methods of the `OpenFileDialog` class is `ShowDialog`. As the name suggests, its purpose is to show the Open dialog box. You can call the `ShowDialog` method by the following code, which starts with the name of the object (`dlgOpen`), followed by a period separating the object name from the method name (`ShowDialog`), followed by empty parentheses (because this method has no parameters):

```
dlgOpen.ShowDialog()
```

Let's test this code in the Text Editor project. Add a button to the form named `btnRead` with the Text property `Read`. Create the following Click event procedure for this button:

```
Private Sub btnRead_Click (ByVal sender As Object, _  
    ByVal e As System.EventArgs) Handles btnRead.Click  
    dlgOpen.ShowDialog()  
End Sub
```

When you run the project and click the `Read` button, the `OpenFileDialog` control will appear, similar to Figure 13-1. The `OpenFileDialog` control is modal, meaning your application cannot continue until the user closes the Open dialog box by clicking one of its two buttons, either `Open` (after selecting a file) or `Cancel`.

Determining Whether Open or Cancel Has Been Chosen

Although choosing either the `Open` or `Cancel` button will close the Open dialog box, it is important to know which button was chosen. If the `Open` button was chosen, we would want our code to open the selected file. However, if the `Cancel` button was chosen, we would not want our code to attempt to open a file because no file was selected.

From the code we have written so far, you can't tell whether the `Open` or `Cancel` button was chosen. Now we will add to the code so we can determine which button was chosen.

In addition to displaying the `OpenFileDialog` control, the `ShowDialog` method also returns a `DialogResult`. The `DialogResult` was discussed in Chapter 10 in connection with dialog forms. As discussed there, the value of the `DialogResult` that is returned by the `ShowDialog` method corresponds to the button the user selected to close the dialog box. For example, if the user chose the `OK` button, the value returned by the `ShowDialog` method is `DialogResult.OK`. However, if the user chose the `Cancel` button, the value returned by the `ShowDialog` method is `DialogResult.Cancel`.

The Open dialog box has an `Open` button instead of an `OK` button, but the `DialogResult` that corresponds to the user's choice of the `Open` button still is

DialogResult.OK. Not surprisingly, the DialogResult is DialogResult.Cancel if the user instead chose the Cancel button to close the Open dialog box.

Here is the syntax for using the return value of the ShowDialog method to determine whether the user chose the Open or Cancel button:

```
Dim dr As DialogResult
dr = dlgOpen.ShowDialog()
If dr = DialogResult.OK Then
    ' Open button was clicked
Else
    'Cancel button was clicked
End If
```

This first statement creates a DialogResult variable because that is the data type returned by the ShowDialog method. The second statement calls the ShowDialog method and assigns its return value to the DialogResult variable we created in the first statement. The following If/Else statement checks to see if the value of the DialogResult variable is DialogResult.OK. If it is, the Open button was clicked. Otherwise, the Cancel button was clicked.

Accordingly, modify the code in the Click event procedure of the Read button so it reads as follows:

```
Private Sub btnRead_Click (ByVal sender As Object, _
ByVal e As System.EventArgs) Handles btnRead.Click
    Dim dr As DialogResult
    dr = dlgOpen.ShowDialog()
    If dr = DialogResult.OK Then
        MessageBox.Show("Open button was clicked")
    End If
End Sub
```

Run the project. Click the Read button to display the Open dialog box. Select a file and click the Open button. The message box will display that the Open button was clicked. Close the message box. Click the Read button again to redisplay the Open dialog box. This time click the Cancel button. No message box will display, indicating that the Cancel button was clicked.

Identifying the File to Open

We have made progress! We can now determine through code whether the user chose the Open or Cancel button. The next step is to determine the name of the file the user chose if they selected the Open button, because we need that name to know which file to open.

The `OpenFileDialog` class has a `FileName` property whose value is a string containing the path to and name of the file selected in the file dialog box. For example, if we chose the file `data.txt` that is in the `C:\temp` directory, the `FileName` property would be `C:\temp\data.txt`.

Usually you are interested in the `FileName` property only if the user chose the Open button. If the user chose the Cancel button instead, the `FileName` property is an empty string.

Modify the code in the Click event procedure of the Read button so it reads as follows:

```
Private Sub btnRead_Click (ByVal sender As Object, _  
    ByVal e As System.EventArgs) Handles btnRead.Click  
    Dim dr As DialogResult  
    dr = dlgOpen.ShowDialog()  
    If dr = DialogResult.OK Then  
        MessageBox.Show(dlgOpen.FileName)  
    End If  
End Sub
```

Run the project. Click the Read button to display the Open dialog box. Select a file and click the Open button. The message box will display the path to and name of the file. You can now close the message box, and then the form.

SaveFileDialog Control

You use a `SaveFileDialog` control to add to your application the ability to save files using the built-in Save dialog box, which is shown in Figure 13-3.

NOTE *The Save dialog box often is titled “Save As” rather than “Save,” as in Figure 13-3. The title depends on, among other factors, if the contents are being saved to a different file than the one opened, or whether the file is being saved for the first time. The discussion in this chapter about the Save dialog box applies equally to the Save As dialog box.*

Add a `SaveFileDialog` control to your form, as you did the `OpenFileDialog` control earlier in this chapter. Name the `SaveFileDialog` control `dlgSave`. You don’t need to change any of this control’s other default properties.

The `SaveFileDialog` control, like the `OpenFileDialog` control, is modal, meaning your application cannot continue until the user closes the Save dialog by clicking one of its two buttons, either Save or Cancel.

Once you have learned how to use an `OpenFileDialog` control, using the `SaveFileDialog` control is easy. The reason is the `ShowDialog` method, the `DialogResult` return value, and the `FileName` property work the same way with a

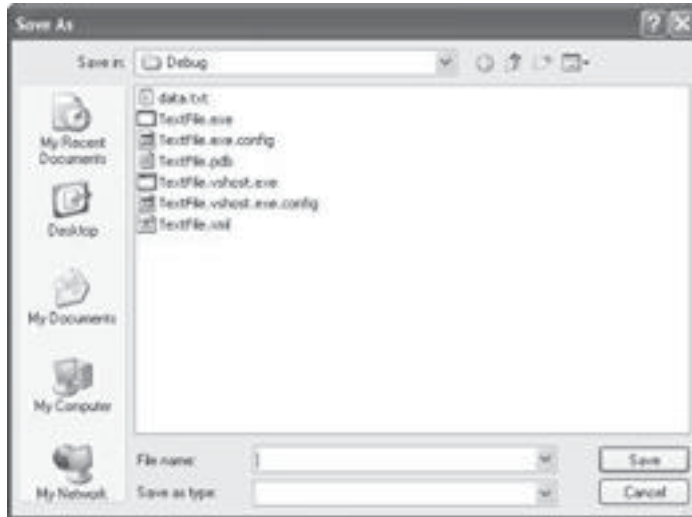


Figure 13-3 Save dialog box

SaveFileDialog control as they do with an OpenFileDialog control. The DialogResult returned by clicking the Save button is DialogResult.OK, just as is the case with clicking the Open button in the OpenFileDialog control.

Let's test this by adding another Button control to the form in the Text Editor project. Name this button btnWrite with the Text property Write. Then create the following Click event procedure for the button:

```
Private Sub btnWrite_Click (ByVal sender As Object, _
ByVal e As System.EventArgs) Handles btnWrite.Click
    Dim dr As DialogResult
    dr = dlgSave.ShowDialog()
    If dr = DialogResult.OK Then
        MessageBox.Show(dlgSave.FileName)
    End If
End Sub
```

Run the project. Click the Write button to display the Save dialog box. Select a file and click the Save button. A message box will display the path to and the name of the file. Another message box always will advise you that the file already exists and ask you if you want to replace it. Answer Yes to close the warning message box (don't worry, the file will not be replaced). The Save dialog box will close. Next, click the Write button to display the Save dialog box again. This time click the Cancel button. No message box will display, indicating that the Cancel button was clicked. Then close the form to end the application.

Reading from a Text File

I am always telling my students that the best way to learn computer programming is to write programs. Therefore, you will learn in this section how to display in the text box in the Text Editor project the contents of a text file selected in an Open dialog box. When we are finished writing code, clicking the “Read” button will display in the TextBox control the contents of a text file. Figure 13-4 shows how the application will appear after the Read button is clicked and the contents of a text file are displayed in the TextBox control.

Conversely, in the next section you will further enhance the project so that when you click the “Write” button, the application will write to the text file the contents of the TextBox control. Thus, if I make any changes to the text of the TextBox control and click the Write button, the text file will be updated with those changes.

StreamReader Class

We will use the StreamReader class to read from the text file. The word “stream” refers to a stream of data, moving from one place to another (in this case, from a text file to your application). The word “reader” means the file is being read. As you might now guess, when we want to write to the file, we will use the StreamWriter class.

To use the StreamReader class, we first will declare a variable of that data type:

```
Dim readerVar As IO.StreamReader
```

The term “IO” precedes StreamReader or else the compiler will complain that the term “StreamReader” is not defined. The reason is that the StreamReader class is part of the System.IO namespace.



Figure 13-4 Application displaying the contents of a text file

Importing the System.IO Namespace

The compiler will not look in the System.IO namespace unless we tell it to. One way to tell the compiler to look in the System.IO namespace is to precede StreamReader with System.IO.

There is an easier way to tell the compiler to look in the System.IO namespace. At the top of the code module, above Public Class Form1, type the following:

```
Imports System.IO
```

Therefore, your code will start with

```
Imports System.IO  
Public Class Form1
```

Including this one Imports statement means that you don't have to precede StreamReader (or StreamWriter) with System.IO each time you use that term in your code. Now you can declare the StreamReader variable readerVar in the Click event procedure of the Read button without preceding StreamReader with IO, as shown here:

```
Dim readerVar As StreamReader
```

Revise the code in your Read button Click event procedure to appear as follows:

```
Private Sub btnRead_Click (ByVal sender As Object, _  
ByVal e As System.EventArgs) Handles btnRead.Click  
    Dim readerVar As StreamReader  
    Dim dr As DialogResult  
    dr = dlgOpen.ShowDialog()  
    If dr = DialogResult.OK Then  
        MessageBox.Show(dlgOpen.FileName)  
    End If  
End Sub
```

Instantiating a StreamReader Variable

Although we have created the StreamReader variable readerVar, right now that variable does not relate to any text file. Therefore, the next step is to connect the StreamReader variable readerVar to the text file we want to read. This process is known as “instantiating the variable.”

We will instantiate the StreamReader variable with the following statement:

```
readerVar = New StreamReader(dlgOpen.FileName)
```

This line of code will replace the code that showed the message box, MessageBox.Show(dlgOpen.FileName), because the message box was for illustration and we

are now actually about to open the selected file for reading rather than just display its path and name.

Thus, so far your Read button Click event procedure should appear as follows:

```
Private Sub btnRead_Click (ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles btnRead.Click
    Dim readerVar As StreamReader
    Dim dr As DialogResult
    dr = dlgOpen.ShowDialog()
    If dr = DialogResult.OK Then
        readerVar = New StreamReader(dlgOpen.FileName)
    End If
End Sub
```

Now let's take a careful look at the statement we've just added, starting from the right side of the assignment statement.

The `New` keyword is used to create a new `StreamReader` instance that points to the text file to be read. The term "StreamReader" in the statement `New StreamReader(dlgOpen.FileName)` indicates the type of instance being created. When the name of the function (here, `StreamReader`) is the same as the name of a class (also `StreamReader`), as here, it is called a *constructor*. The constructor is used to "construct" the new instance.

The constructor in this code example takes one argument: the name of the file to be read. That file name is obtained from the `FileName` property of the Open dialog box.

The right side of the assignment operator returns the new instance, which then is assigned to the `StreamReader` variable `readerVar` on the left side of the assignment operator. Now the `StreamReader` variable `readerVar` is connected to the text file we want to read.

Reading the Text File into the TextBox

The `StreamReader` class has a `ReadToEnd` method, which returns a string representing the entire text of the text file. We then assign that string to the `Text` property of `txtEdit` so that the text of the `TextBox` control will display the entire text of the text file. Accordingly, add the following statement to your Read button Click event procedure:

```
txtEdit.Text = readerVar.ReadToEnd()
```

Your Read button Click event procedure now should read as follows:

```
Private Sub btnRead_Click (ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles btnRead.Click
    Dim readerVar As StreamReader
    Dim dr As DialogResult
    dr = dlgSave.ShowDialog()
```

```
If dr = DialogResult.OK Then
    readerVar = New StreamReader(dlgOpen.FileName)
    txtEdit.Text = readerVar.ReadToEnd()
End If
End Sub
```

The `StreamReader` class has other methods that are alternatives to `ReadToEnd`. The `Read` method reads a specified number of characters, and the `ReadLine` method reads a line. For example, if you want to load the data one line at a time into a row of a control, the `ReadLine` method might be a logical choice.

Closing the Text File

Once we have read the entire contents of the text file, there is no further need to read from it. Therefore, we should close the text file for reading. The `StreamReader` class has a `Close` method to accomplish this. Accordingly, add the following line of code to close the text file for reading:

```
readerVar.Close()
```

This completes the `Read` button `Click` event procedure, which now should read as follows:

```
Private Sub btnRead_Click (ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles btnRead.Click
    Dim readerVar As StreamReader
    dr = dlgOpen.ShowDialog()
    If dr = DialogResult.OK Then
        readerVar = New StreamReader(dlgOpen.FileName)
        txtEdit.Text = readerVar.ReadToEnd
        readerVar.Close()
    End If
End Sub
```

Closing the text file for reading frees system resources, specifically memory. This is important. Memory is required to keep a file open for reading (or writing). When you don't need to keep the file open anymore, you should give the memory back to the operating system.

By analogy, a library would run out of books if patrons checked out books but never returned them when they were finished reading the books. Similarly, your computer only has so much available memory for applications (some memory is needed by the operating system itself). If applications don't return memory after checking it out, the operating system eventually will run out of memory. The consequence of the operating system running out of available memory for applications often is a general protection fault or illegal exception, bringing the user's work to a crashing halt.

Additionally, later in this chapter you may be writing to the same text file that you read. Trying to open a file for writing that already is open for reading may cause problems, which can be avoided by closing the file first before reopening it for another purpose.

Run the project. Click the Read button. Use the resulting Open dialog box to select and open a text file. The contents of that text file should be displayed in the text box. You can then close the application.

Writing to a Text File

The next step in enhancing the Text Editor project will be to write to the text file by copying the contents of the text box to the text file. The code to do this will be in the Click event procedure of the Write button.

StreamWriter Class

We will now change the code previously in this chapter for the Click event procedure for the Write button by replacing the code displaying the message box with the following code:

```
Dim writerVar As StreamWriter
writerVar = New StreamWriter(dlgSave.FileName, False)
```

The code for the Click event procedure of the Write button should now look like this:

```
Private Sub btnWrite_Click (ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles btnWrite.Click
    Dim dr As DialogResult
    dr = dlgSave.ShowDialog()
    If dr = DialogResult.OK Then
        Dim writerVar As StreamWriter
        writerVar = New StreamWriter _
            (dlgSave.FileName, False)
    End If
End Sub
```

The two lines of code we just added may look familiar from the code we wrote earlier in this chapter for the StreamReader. There we declared a StreamReader variable and then instantiated that variable using the StreamReader constructor to read a text file. Here we are declaring a StreamWriter variable and then instantiating that variable using the StreamWriter constructor to write to a text file. As the name suggests, the StreamWriter class is used when writing to a text file.

The first argument of the `StreamWriter` constructor is the name of the text file. This is the same as the first argument of the `StreamReader` constructor. However, the `StreamWriter` constructor has an additional, second argument.

NOTE *The `StreamWriter` constructor, like the `Show` method of the `MessageBox` class, is overloaded, which means that it may be called with a different number of arguments.*

The data type of the second argument of the `StreamWriter` constructor is `Boolean`. The value of this second argument is `True` if you want to add to the existing contents of the text file, and `False` if instead you want to overwrite the existing contents of the text file.

In this project, we want to overwrite rather than add to the existing contents of the text file. Accordingly, the value of the second argument is `False`.

If you instead wanted to add to the existing contents of the file, you would use `True` instead of `False` as the second argument of the `StreamWriter` constructor. One example would be a log file, which logs events or problems. Normally you would want to add a new event or problem to the prior list, not erase the prior list in the process.

Writing from the TextBox to the Text File

The `StreamWriter` class has a `Write` method that writes the contents of its argument to the text file at which the `StreamReader` instance is targeted. In this application, we want to write the contents of the text box to the text file. Thus, the argument is the `Text` property of the `TextBox` control. Accordingly, add the following code to the `Click` event of the `Write` button:

```
writerVar.Write(txtEdit.Text)
```

The code for the `Click` event procedure of the `Write` button should now look like this:

```
Private Sub btnWrite_Click (ByVal sender As Object, _  
ByVal e As System.EventArgs) Handles btnWrite.Click  
    Dim dr As DialogResult  
    dr = dlgSave.ShowDialog()  
    If dr = DialogResult.OK Then  
        Dim writerVar As StreamWriter  
        writerVar = New StreamWriter _  
            (dlgSave.FileName, False)  
        writerVar.Write(txtEdit.Text)  
    End If  
End Sub
```

Closing the Text File

We are now finished writing to the text file. Accordingly, we should close the text file for writing, as we closed the text file for reading earlier in this chapter. Therefore, add the following statement to the Click event of the Write button:

```
writerVar.Close()
```

The completed code for the Click event procedure of the Write button should now look like this:

```
Private Sub btnWrite_Click (ByVal sender As Object, _  
ByVal e As System.EventArgs) Handles btnWrite.Click  
    Dim dr As DialogResult  
    dr = dlgSave.ShowDialog()  
    If dr = DialogResult.OK Then  
        Dim writerVar As StreamWriter  
        writerVar = New StreamWriter _  
            (dlgSave.FileName, False)  
        writerVar.Write(txtEdit.Text)  
        writerVar.Close()  
    End If  
End Sub
```

CAUTION *Your program may make changes to your text file, and you don't want those changes to cause any problems on your computer. Accordingly, before you test this project, create a text file using Notepad or another plain text editor and type in whatever contents you would like. However, don't use Microsoft Word or a comparable word processing program to create the text file because these programs include formatting characters as well as text.*

Run the project. Click the Read button. Use the resulting Open dialog box to select and open the text file you created. The contents of that text file should be displayed in the text box. Then, make changes in the text box. When done making changes in the text box, click the Write button. When the Save dialog box displays, find and choose the text file you created and then click the Save button. You may see a message box that informs you that the file you are saving to already exists and asking you if you want to replace it. Click the Yes button.

Run your application again and display the text file. The text should show the changes you made when you first ran the application.

This application is not yet ready for prime time. For example, we should disable the Write button until a file is opened with the Read button. We also should create File | Open and File | Save menu items and link their Click events to the Click events of the Read and Write buttons. You may wish to try to implement these enhancements. Nevertheless, this project is useful in demonstrating how to read from and write to a text file.

Conclusion

In this chapter, you learned how to add to your program Open and Save dialog boxes that sophisticated programs such as Microsoft Word have. The Open dialog box is a control of the `OpenFileDialog` class. Similarly, the Save dialog box is a control of the `SaveFileDialog` class. You use the `ShowDialog` method to display each dialog, and the `DialogResult` property to determine if the user chose the dialog box's Open or Save button, or instead the Cancel button. If the user chose the Open (or Save) button, you use the `FileName` property to retrieve the file name chosen by the user from the dialog box.

In this chapter, you also learned how to read from a text file using the `StreamReader` class and to write to a text file using the `StreamWriter` class. Although text files may not seem as advanced as databases, one advantage text files have over databases is that they are universally understood by applications, whereas databases require specialized software.

However, databases also have their advantages, so the next chapter will be about them.

Quiz

1. The Open dialog box is a control of which class?
2. What method do you use to show an Open dialog box?
3. What is the return value of showing an Open dialog box?
4. What is the property of the `OpenFileDialog` class whose value is the file chosen by the user in an Open dialog box?
5. The Save dialog box is a control of which class?



6. What method do you use to show a Save dialog box?
7. What is the return value of showing a Save dialog box?
8. What is the property of the SaveFileDialog class whose value is the name of the file to be saved?
9. What class may you use to read from a text file?
10. What class may you use to write to a text file?

Databases

Up until now, we have saved data in a text file. But text files have their limitations. One limitation is that it is difficult to quickly retrieve specific data in a text file. There's usually no alternative to searching the text file from beginning to end, which can take a long time if the text file contains a lot of data.

Another limitation of a text file is its inability to link different but related data. For example, a store may have both a list of customers and a list of orders. Because the orders come from customers, the two different lists are related. But with a text file, there's no easy way to link an order in one list with a customer in another list.

A database does not have these limitations—you can quickly retrieve specific data using keys and indexes, and you can easily link different data.

Although there are many types of databases, fundamentally these different database types share a number of common characteristics. Accordingly, you will be able to apply what you learn here to different types of databases.

This chapter will get you started with databases. However, I'm not going to start with a dry theoretical discussion of what a database is because that information can be a little abstract if you haven't first spent some time working with one. So let's roll up our sleeves (figuratively, of course) and get started working with a database.

Installing the Database

Databases come in different formats. Microsoft Access, Microsoft SQL Server, and Oracle are among the most common, but there are many others, each with their advantages, disadvantages, followers, and detractors.

I'll be using a Microsoft Access database in this chapter solely because I believe my readers are more likely to have Microsoft Access than other database products such as Microsoft SQL Server and Oracle. Additionally, it is easier to get started using Microsoft Access than with most other database products. However, you will be able to apply what you learn here to other database formats such as Microsoft SQL Server and Oracle.

Obtaining the Northwind Traders Database

We will be working with the Northwind Traders database. It is a Microsoft Access database and is on the installation CD for Microsoft Access.

However, you can use the Northwind Traders database with Visual Basic 2005 without having Microsoft Access. Microsoft permits you to download, free of charge, a version of this sample database for Access 2000. This version also will work if you have Access XP or 2003.

The download link at the time of this book is

<http://www.microsoft.com/downloads/details.aspx?FamilyID=c6661372-8dbe-422b-8676-c632d66c529c&displaylang=en>

This link may change, particularly when Microsoft periodically reorganizes its website. In case you need to do a search, the title of the article is "Access 2000 Tutorial: Northwind Traders Sample Database."

Installing the Northwind Traders Database

The name of the installation file is Nwind.exe. Once you download this file onto your hard drive, double-click it to start the installation process. The installation program will ask you to agree to a license to use the database and then ask where you want to save the database. Save it wherever you wish on the hard drive; just remember where you saved it.

The saved database may have the name Nwind.mdb or Northwind.mdb. The .mdb extension is an abbreviation of "Microsoft database" and is used for Access databases.

Connecting to the Database

If you have Access, you can view the Northwind Traders database from that application.

You can also view the Northwind Traders database via Visual Basic 2005. You don't need to open or create a Windows application. However, you first need to connect Visual Basic 2005 to the database.

To start the process of connecting Visual Basic 2005 to the Northwind Traders database, choose the Tools | Connect to Database menu command. This will display the Choose Data Source dialog box shown in Figure 14-1.

As Figure 14-1 shows, the upper pane of the Choose Data Source dialog box lists different database formats, such as Access, SQL Server, and Oracle. Because Northwind Traders is an Access database, choose Microsoft Access Database File. Figure 14-2 shows the Choose Data Source dialog box after you choose Microsoft Access Database File.

As Figure 14-2 shows, the drop-down box below the upper pane, blank in Figure 14-1, now lists the one available data provider: .NET Framework Data Provider for OLE DB. A data provider is a code component that is used by your application to connect to a specific database format. There are many database formats, so there are many providers, at least one for each database format supported by the .NET Framework. The .NET Framework may have several alternative data providers for some database formats, but it just has one, the .NET Framework Data Provider for OLE DB, for the Microsoft Access database format.

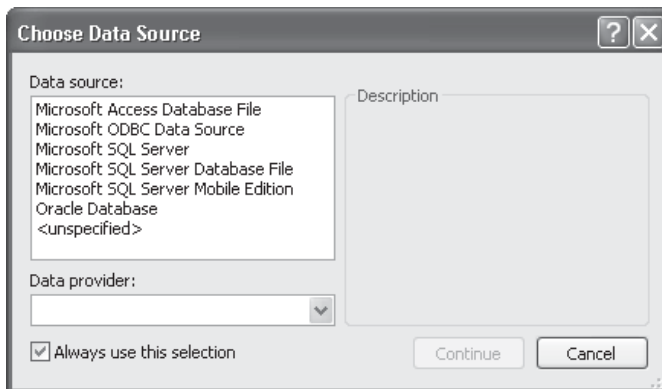


Figure 14-1 Choose Data Source dialog box

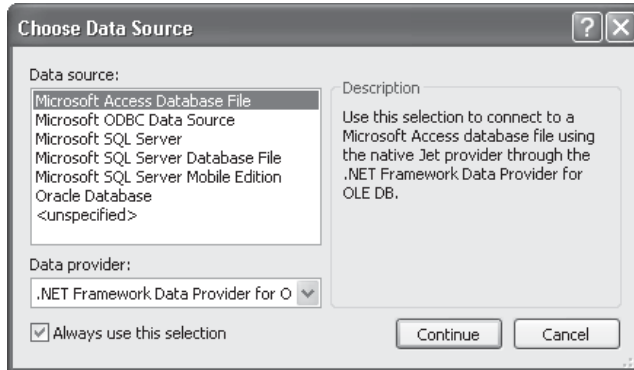


Figure 14-2 Choose Data Source dialog box after the data source is selected

As Figure 14-2 also shows, the Description area to the right of the upper pane, blank in Figure 14-1, now contains the following text: “Use this selection to connect to a Microsoft Access database file using the native Jet provider through the .NET Framework Data Provider for OLE DB.” The reason for the term “Jet” is that Microsoft Access uses the Jet database engine.

Finally, Figure 14-2 shows that once you have selected a data source and a data provider, the Continue button, disabled in Figure 14-1, now is enabled.

Click the Continue button. This will display the Add Connection dialog box shown in Figure 14-3.

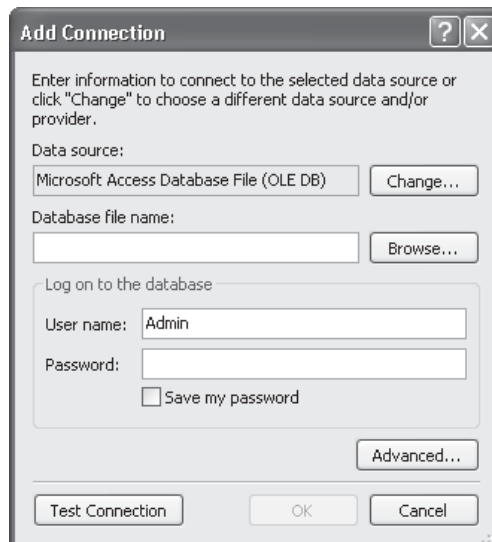


Figure 14-3 Add Connection dialog box



Figure 14-4 Add Connection dialog box after the database is selected

Use the **Browse** button to find and choose the `nwind.mdb` file you saved on your hard drive when you installed the Northwind Traders database. Once you have done this, as shown in Figure 14-4, the path to and name of the database should appear in the Database File Name text box.

NOTE *You don't need to worry about the user name and password in the Add Connection dialog box, unless you assigned a name and password to the database, which you don't need to do. This may be an issue with other database formats, but it's not an issue with Microsoft Access.*

The next step is to test the connection. Click the **Test Connection** button. A message box stating “Test Connection Succeeded” should display, as in Figure 14-5.

Click the **OK** button. This saves the changes you made and closes the Add Connection dialog box.

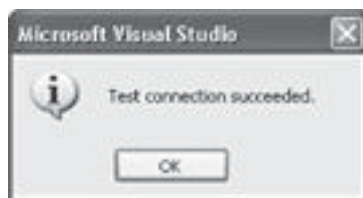


Figure 14-5 Test connection succeeded.

Using Server Explorer

If you have Microsoft Access, you can use it to view the Northwind Traders database. If you don't, Visual Basic 2005 has a tool called Server Explorer that permits you to view and make changes to databases on your computer or on any other computer to which you have network access and permissions.

Indeed, you should learn how to use Server Explorer even if you have Microsoft Access on your computer. First, you may find yourself working at another computer that doesn't have Microsoft Access. Second and perhaps more important, when you're working with other database formats such as SQL Server or Oracle, you won't be able to use Microsoft Access.

You can display Server Explorer using the View | Server Explorer menu command. You don't need to first open or create a Windows application. Figure 14-6 shows Server Explorer after the Data Connections node was expanded by clicking the + sign to its left.



Figure 14-6 Server Explorer

NOTE *Server Explorer on your machine will likely have different content than what's shown in Figure 14-6. For example, PCKlub866 is listed under the Servers node because that happens to be the name of the computer I used.*

The node underneath the Data Connections node should list the path and file name of the Microsoft Access database to which we just created a connection in the previous section “Connecting to the Database.”

Exploring the Database

Click the + sign next to the Microsoft Access database under the Data Connections node. As Figure 14-7 shows, four nodes appear: Tables, Views, Stored Procedures, and Functions.

A *table* is a collection of data on a particular subject. In this chapter, we'll be discussing a particular table, Customers. The Northwind Traders database has other tables, too, including those listing employees, products, orders, suppliers, and shippers.



Figure 14-7 Server Explorer listing Tables, Views, Stored Procedures, and Functions

A *view* is a collection of data, often obtained from more than one table. Examples of views in the Northwind Traders database include “Product Sales for 1995” and “Ten Most Expensive Products.”

A *stored procedure* and a *function* each is generally a code component that generates a predefined subset of the data. Examples of stored procedures in the Northwind Traders database include “Alphabetical List of Products” and “Summary of Sales by Year.” Examples of functions in the Northwind Traders database include “Sales by Year.”

Exploring the Customers Table

Click the + sign next to the Tables node. As Figure 14-8 shows, this displays the various tables in the Northwind Traders database.

Click the + sign next to the Customers table. As Figure 14-9 shows, this displays the various fields of the Customers table.



Figure 14-8 Server Explorer listing Tables



Figure 14-9 Fields of Customers table

Right-click the Customers table node and choose Show Table Data from the shortcut menu. As Figure 14-10 shows, the data in the Customers table then will be displayed.

As Figure 14-10 shows, the data in the Customers table is displayed in rows and columns. Each column, or field, represents a different piece of information, such as a name, title, or address. Each row, or record, concerns one customer. Together, the rows and columns provide information, such as the name, title, and address of each customer.

Different tables have different fields and a different number of records. Additionally, the fields are not always of a String data type, but instead may be of another data type, such as Integer or Boolean. The one thing tables have in common is that they're composed of fields (columns) and records (rows).

CustomerID	CompanyName	ContactName	ContactTitle	Address	City	Region	PostalCode
BONAP	Bon app!	Laurence Lebhan	Owner	12, rue des Bouc...	Marseille	NAE1	13000
BOTFM	Buttons-Gollar M...	Elizabeth Lincoln	Accounting Man...	23 Tsvessien Bl...	Tsvessien	BC	32F 894
BSEVY	B's Beverages	Victoria Ashworth	Sales Represent...	Fountainery Circus	London	NAE1	EC2 1NF
CACTU	Cactus Comidas ...	Pablo Simpson	Sales Agent	Centro 333	Buenos Aires	NAE1	1010
CENTC	Centro comercial ...	Francisco Chang	Marketing Manager	Siemas de Grana...	México D.F.	NAE1	05022
CHOPS	Chop-suey Chinese	Yang Wang	Owner	Hauptstr. 29	Bern	NAE1	3012
COMPE	Comércio Mineiro	Pedro Afonso	Sales Associate	Av. dos Lusitades...	São Paulo	SP	05432-040
CONSH	Consolidated Hol...	Elizabeth Brown	Sales Represent...	Berkeley Garden...	London	NAE1	W05 4LT
DRACD	Drachentakt Del...	Sven Ottlieb	Order Administrator	Waldenweg 21	Wetzlar	NAE1	52066
DUMON	Du monde entier	Jeanine Labruno	Owner	67, rue des Orp...	Nantes	NAE1	44000
EASTC	Eastern Connection	Ann Devon	Sales Agent	35 King George	London	NAE1	W03 4PW
ERRSH	Ersel Handel	Roland Mendel	Sales Manager	Kirchgasse 6	Graz	NAE1	8010
FAMSA	Familia Arzobispo	Aris Cruz	Marketing Assit...	Rua Orós, 92	São Paulo	SP	05442-030
FESSA	FESSA-Fabrica In...	Diego Roel	Accounting Man...	C/ Morabarral, 06	Madrid	NAE1	28004
FOLIG	Folies gourmandes	Martine Rancil	Assistant Sales ...	104, chaussée d...	Lille	NAE1	59000
FOLKO	Folk och Fä HE	Maria Larsson	Owner	Årsgatan 24	Briekle	NAE1	S-044 67
FRANK	Frankenversand	Peter Franken	Marketing Manager	Berliner Platz 43	München	NAE1	80805
FRANI	France restauration	Carine Schmitt	Marketing Manager	54, rue Royale	Nantes	NAE1	44000
FRANS	France restauration	Carine Schmitt	Marketing Manager	54, rue Royale	Nantes	NAE1	44000

Figure 14-10 Data in Customers table

Database Project

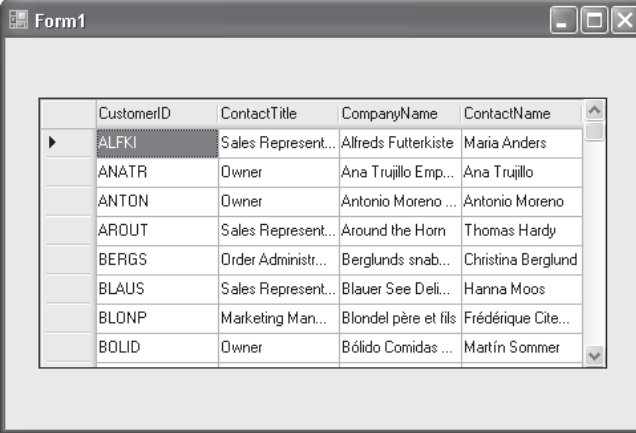
As you have heard me say several times already in this book, the best way to learn programming is to write programs. So let's put that saying into practice once again.

What the Project Does

This project, when finished, will, when the application starts up, fill a DataGridView control with data from four fields of the Customers table: CustomerID, ContactTitle, CompanyName, and ContactName. Figure 14-11 shows the project in action.

Creating the Form

Create a new Windows application. Add two controls to the default form.



CustomerID	ContactTitle	CompanyName	ContactName
ALFKI	Sales Represent...	Alfreds Futterkiste	Maria Anders
ANATR	Owner	Ana Trujillo Emp...	Ana Trujillo
ANTON	Owner	Antonio Moreno ...	Antonio Moreno
AROUT	Sales Represent...	Around the Horn	Thomas Hardy
BERGS	Order Administr...	Berglunds snab...	Christina Berglund
BLAUS	Sales Represent...	Blauer See Deli...	Hanna Moos
BLONP	Marketing Man...	Blondel père et fils	Frédérique Cite...
BOLID	Owner	Bólido Comidas ...	Martín Sommer

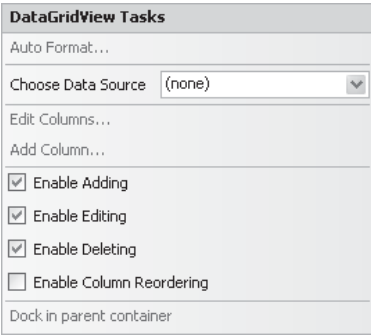
Figure 14-11 DataGridView control filled with data from Customers table

The first control is an OpenFileDialog control. You learned about this control in Chapter 13. Name this control `dlgOpen` and delete any value in its `FileName` property. You do not need to change any of its other default properties.

The second control is a DataGridView control. This control displays data in a row and column format, much like the Customers table shown in Figure 14-10, or a spreadsheet.

When you add the DataGridView control, a DataGridView Tasks pane displays, as shown in Figure 14-12.

You may accept the default values in this pane. However, center the DataGridView control in your form and rename it `dgvData`.



DataGridView Tasks	
Auto Format...	
Choose Data Source	(none)
Edit Columns...	
Add Column...	
<input checked="" type="checkbox"/> Enable Adding	
<input checked="" type="checkbox"/> Enable Editing	
<input checked="" type="checkbox"/> Enable Deleting	
<input type="checkbox"/> Enable Column Reordering	
Dock in parent container	

Figure 14-12 DataGridView Tasks pane

Importing Data Namespaces

The code components used for database access are organized in the .NET class library under the name ADO.NET. You've probably already figured out the ".NET" portion of that name. ADO was an acronym for ActiveX Data Objects, a Microsoft data-access technology that preceded ADO.NET.

Several ADO.NET classes, which we will use in this chapter, are part of the System.Data.OleDb namespace. As you may remember from previous chapters, the .NET library is organized in a hierarchal structure, each branch with its own namespace. System is a top-level namespace. Data is one of several namespaces belonging to System, and OleDb is one of several namespaces belonging to System.Data.

NOTE *There are other namespaces supporting other database types, such as OracleClient for Oracle databases and SqlClient for SQL Server databases.*

Thus, the OleDbConnection class we will be using in the next section technically is not just the OleDbConnection class but instead the System.Data.OleDb.OleDbConnection class. However, typing a System.Data.OleDb prefix before every reference to OleDbConnection or another ADO.NET class can quickly become a pain.

Fortunately, you can avoid having to prefix every reference to an ADO.NET class with System.Data.OleDb by using an Imports System.Data.OleDb statement before your class declaration. While you are at it, also import the System.Data namespace, because that namespace also will come in handy later. Thus, assuming the class name of your form is Form1, the first three lines of code will be

```
Imports System.Data
Imports System.Data.OleDb
Public Class Form1
```

If the compiler does not recognize the root System.Data namespace, you may need to add a reference to the assembly that contains the namespace. Choose Add Reference from the Project menu to display the Add Reference dialog box shown in Figure 14-13.

Choose System.Data from the list and click OK. Then the compiler will recognize the root System.Data namespace.

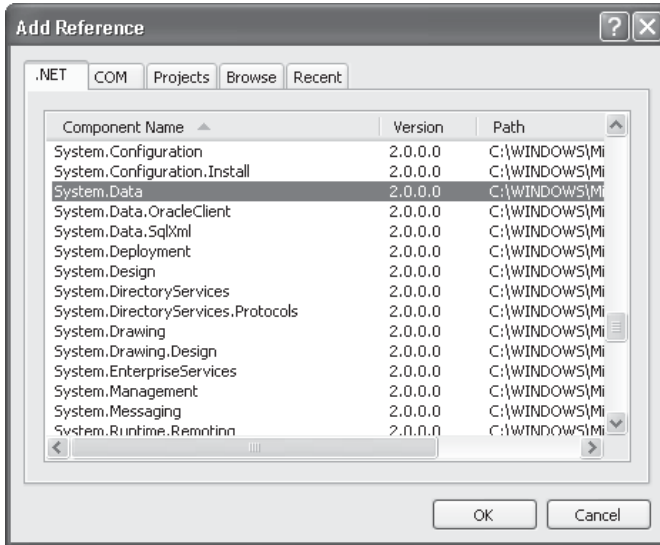


Figure 14-13 Add Reference dialog box

Creating a Connection

Your application will be giving commands to the database to retrieve certain data. But before it can do so, your application needs to have a connection with the database.

Persistent Connection vs. Disconnected Application

Although an application needs to have a connection to a database to retrieve or change data, there is more than one way to design this connection. One alternative is to create a single connection that remains active until the application ends. This is called a *persistent connection*.

The alternative is to create a connection only to retrieve data, end the connection, make changes to a local copy of the data while disconnected from the database, and connect back to the database only when necessary to synchronize these changes with the database. This is called a *disconnected application* because most of the time, the application is disconnected from the database.

As with most choices in life, there are tradeoffs between a persistent connection and a disconnected application. In general, Windows applications are more likely to use persistent connections, whereas web applications are more likely to be disconnected applications, but this is only a generalization, of course.

Because we are writing a Windows application, we will use a persistent connection.

OleDbConnection Class

The OleDbConnection class represents a connection to a data source. The following line of code not only declares an OleDbConnection variable, but also instantiates it:

```
Dim myConn As New OleDbConnection
```

As explained in previous chapters, the term “instantiate” means to create a new instance (in this case, a new connection). This instantiation is performed by using the New keyword when declaring the OleDbConnection variable.

ConnectionString Property

The OleDbConnection class has a ConnectionString property. This property includes the provider being used and the path to and the name of the data source file.

The provider is “Microsoft.Jet.OLEDB.4.0.” As mentioned in the earlier section “Connecting to the Database,” the reason for the term “Jet” is that Microsoft Access uses the Jet database engine. Additionally, as the Description area of the Choose Data Source dialog box shown in Figure 14-2 reflects, the connection to a Microsoft Access database uses the native Jet provider through the .NET Framework Data Provider for OLE DB. That native Jet provider is Microsoft.Jet.OLEDB.4.0.

We will obtain the path to and the name of the data source file through the OpenFileDialog control and its FileName property.

In this project, all the code will be written in the Load event of the form. Write the following code:

```
Private Sub Form1_Load (ByVal sender As System.Object, _  
ByVal e As System.EventArgs) Handles Me.Load  
    Dim myConn As New OleDbConnection  
    Dim dr As DialogResult  
    dr = dlgOpen.ShowDialog()  
    If dr = DialogResult.OK Then  
        Dim strFile As String = dlgOpen.FileName  
        myConn.ConnectionString = "Provider=" & _  
            "Microsoft.Jet.OLEDB.4.0;Data Source=" & _  
            strFile & ";"  
    End If  
End Sub
```


Opening the Connection

Once you've instantiated an `OleDbConnection` object and created its connection string, you may open the connection to your database using the `OleDbConnection` object's `Open` method:

```
myConn.Open()
```

Accordingly, our code now reads as follows:

```
Private Sub Form1_Load (ByVal sender As System.Object, _  
ByVal e As System.EventArgs) Handles Me.Load  
    Dim myConn As New OleDbConnection  
    Dim dr As DialogResult  
    dr = dlgOpen.ShowDialog()  
    If dr = DialogResult.OK Then  
        Dim strFile As String = dlgOpen.FileName  
        myConn.ConnectionString = "Provider = " & _  
            "Microsoft.Jet.OLEDB.4.0;Data Source=" & _  
            strFile & ";"  
        myConn.Open()  
    End If  
End Sub
```

Creating a Command

Once you establish a connection, you'll next want to execute commands, such as to retrieve data that you want to view. You use an `OleDbCommand` object to execute commands to a database. The `OleDbCommand` class, like the `OleDbConnection` class, is part of the `System.Data.OleDb` namespace.

You instantiate an `OleDbCommand` object similar to how you instantiate an `OleDbConnection` object:

```
Dim myCMD As New OleDbCommand
```

SQL Statement

Commands often are expressed in a SQL statement. SQL, alternatively pronounced as separate letters (*S-Q-L*) or as *sequel*, is an acronym for Structured Query Language. SQL is a standardized language for requesting information from a database.

The following SQL `SELECT` statement retrieves data from the `CustomerID`, `ContactTitle`, `CompanyName`, and `ContactName` fields from the `Customers` table:

```
SELECT CustomerID, ContactTitle, CompanyName, _  
ContactName FROM Customers
```

SELECT is a keyword that indicates the SQL statement retrieves records. The SELECT statement does not change records. Other SQL statements, such as INSERT, UPDATE, and DELETE, do change records, by adding, editing, and deleting, respectively.

The names following the SELECT keyword are the names of table fields. Because there is more than one field, the field names are separated by commas.

FROM is also a keyword. The name following it, Customers, is the name of the table to which the fields belong.

CommandText Property

The OleDbCommand object has a CommandText property whose value may be a SQL statement. Accordingly, we will assign the SQL SELECT statement we discussed in the preceding section to the OleDbCommand object's CommandText property as follows:

```
myCMD.CommandText = "SELECT CustomerID, " & _  
"ContactTitle, CompanyName, ContactName FROM Customers"
```

Note The value of the CommandText property may also be a table name or the name of a stored procedure.

Linking the Command to a Connection

The final step is to link the command to a connection to the database. The OleDbCommand object has a Connection property whose value is the database connection to be used by the command. Accordingly, the following code assigns the existing OleDbConnection variable myConn to the Connection property of the OleDbCommand object:

```
myCMD.Connection = myConn
```

Accordingly, our code now reads like this:

```
Private Sub Form1_Load (ByVal sender As System.Object, _  
ByVal e As System.EventArgs) Handles Me.Load  
    Dim myConn As New OleDbConnection  
    Dim dr As DialogResult  
    dr = dlgOpen.ShowDialog()  
    If dr = DialogResult.OK Then  
        Dim strFile As String = dlgOpen.FileName  
        myConn.ConnectionString = "Provider=" & _
```

```
        "Microsoft.Jet.OLEDB.4.0;Data Source=" & _  
        strFile & ";"  
myConn.Open()  
Dim myCMD As New OleDbCommand  
myCMD.CommandText = "SELECT CustomerID, " & _  
        "ContactTitle, CompanyName, " & _  
        "ContactName FROM Customers"  
myCMD.Connection = myConn  
End If  
End Sub
```

Filling the DataGridView

We now have defined a database connection and command. Here are the remaining tasks:

1. Package that database connection and database command in an OleDbDataAdapter object.
2. Create a DataSet object.
3. Use the OleDbDataAdapter object to fill the DataSet.
4. Use the DataSet to fill the DataGridView.

Creating an OleDbDataAdapter

The OleDbDataAdapter class packages a database connection with a set of data commands.

The first step is to instantiate an OleDbDataAdapter variable, similar to how we previously instantiated the OleDbConnection and OleDbCommand variables:

```
Dim myAdapter As New OleDbDataAdapter
```

The OleDbDataAdapter class has a SelectCommand property whose value is a command that contains a SQL SELECT statement. Accordingly, the following code sets the OleDbDataAdapter variable's SelectCommand property to the OleDbCommand variable we instantiated and configured in the previous section:

```
myAdapter.SelectCommand = myCMD
```

This statement not only connects the OleDbDataAdapter variable to the data command it will use, it also indirectly connects the OleDbDataAdapter variable to the database connection, because the OleDbCommand variable is connected through its Connection property to the OleDbConnection variable.

Accordingly, the code now reads as follows:

```
Private Sub Form1_Load (ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles Me.Load
    Dim myConn As New OleDbConnection
    Dim dr As DialogResult
    dr = dlgOpen.ShowDialog()
    If dr = DialogResult.OK Then
        Dim strFile As String = dlgOpen.FileName
        myConn.ConnectionString = "Provider = " & _
            "Microsoft.Jet.OLEDB.4.0;Data Source=" & _
            strFile & ";"
        myConn.Open()
        Dim myCMD As New OleDbCommand
        myCMD.CommandText = "SELECT CustomerID, " & _
            "ContactTitle, CompanyName, " & _
            "ContactName FROM Customers"
        myCMD.Connection = myConn
        Dim myAdapter As New OleDbDataAdapter
        myAdapter.SelectCommand = myCMD
    End If
End Sub
```

Creating a DataSet

The data used to fill the DataGridView cannot come directly from the hard drive where the database is stored. Instead, an intermediate step is required. The data from the hard drive first must be loaded into memory, or RAM. Then, the data in RAM is loaded into the DataGridView.

NOTE *This approach has advantages. For example, it frees the application from having to exactly replicate the physical data and instead work with subsets, supersets, calculated fields, and so forth.*

A DataSet is a representation of the data (in this case, from several fields of the Customers table, which is stored in RAM).

The DataSet class is part of the System.Data namespace, so you should add an Imports System.Data statement, if you did not do so already earlier in this chapter in the section “Importing Data Namespaces”:

```
Imports System.Data
Imports System.Data.OleDb
Public Class Form1
```

You also need to add a reference to the assembly that contains the namespace System.XML. You do so the same way you added a reference to the assembly that

contains the namespace `System.Data` earlier in this chapter in the section “Importing Data Namespaces”: using the Add Reference dialog box shown in Figure 14-13.

You instantiate a `DataSet` variable via the following code, similar to how we previously instantiated the `OleDbConnection`, `OleDbCommand`, and `OleDbDataAdapter` variables:

```
Dim ds As New DataSet
```

The next steps are to clear and then fill the `DataSet`.

The `DataSet` object has a `Clear` method. This method, as its name suggests, clears the `DataSet` of any leftover contents. There would be no leftover contents here because the code is running on application startup, but often you will need to use the `Clear` method, so it is a good idea to get into the habit of using it.

```
ds.Clear()
```

The `OleDbDataAdapter` object has a `Fill` method. This method, as its name suggests, fills the `DataSet` with its contents, which, once the `DataGridView` is connected to the `DataSet` (as discussed in the next section), then are displayed in the `DataGridView` that is bound to the `DataSet`. The first argument is the `DataSet` to be filled. The second argument is the name of the source table (here, `Customers`).

```
myAdapter.Fill(ds, "Customers")
```

Accordingly, the code now reads as follows:

```
Private Sub Form1_Load (ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Me.Load
    Dim myConn As New OleDbConnection
    Dim dr As DialogResult
    dr = dlgOpen.ShowDialog()
    If dr = DialogResult.OK Then
        Dim strFile As String = dlgOpen.FileName
        myConn.ConnectionString = "Provider = " & _
            "Microsoft.Jet.OLEDB.4.0;Data Source=" & _
            strFile & ";"
        myConn.Open()
        Dim myCMD As New OleDbCommand
        myCMD.CommandText = "SELECT CustomerID, " & _
            "ContactTitle, CompanyName, " & _
            "ContactName FROM Customers"
        myCMD.Connection = myConn
        Dim myAdapter As New OleDbDataAdapter
        myAdapter.SelectCommand = myCMD
        Dim ds As New DataSet
        ds.Clear()
        myAdapter.Fill(ds, "Customers")
    End If
End Sub
```

Connecting the DataGridView to the DataSet

The final step is to connect the DataGridView to the DataSet. This step involves two properties of the DataGridView object: DataSource and DataMember.

The DataSource property is the data source of the data that the DataGridView is displaying. That data source is represented by the DataSet variable ds:

```
dgvData.DataSource = ds
```

The DataMember property is the name of the table (here, Customers) in the data source of the data that the DataGridView is displaying:

```
dgvData.DataMember = "Customers"
```

Accordingly, the completed code now reads like this:

```
Private Sub Form1_Load (ByVal sender As System.Object, _  
ByVal e As System.EventArgs) Handles Me.Load  
    Dim myConn As New OleDbConnection  
    Dim dr As DialogResult  
    dr = dlgOpen.ShowDialog()  
    If dr = DialogResult.OK Then  
        Dim strFile As String = dlgOpen.FileName  
        myConn.ConnectionString = "Provider = " & _  
            "Microsoft.Jet.OLEDB.4.0;Data Source=" & _  
            strFile & "";"  
        myConn.Open()  
        Dim myCMD As New OleDbCommand  
        myCMD.CommandText = "SELECT CustomerID, " & _  
            "ContactTitle, CompanyName, " & _  
            "ContactName FROM Customers"  
        myCMD.Connection = myConn  
        Dim myAdapter As New OleDbDataAdapter  
        myAdapter.SelectCommand = myCMD  
        Dim ds As New DataSet  
        ds.Clear()  
        myAdapter.Fill(ds, "Customers")  
        dgvData.DataSource = ds  
        dgvData.DataMember = "Customers"  
    End If  
End Sub
```

Run the project! The DataGridView control should fill with data, as shown earlier in Figure 14-11.

Conclusion

Text files, which we've used up until now to save data, have several limitations. One limitation is a text file's inability to quickly retrieve specific data. There's usually no alternative to searching the text file from beginning to end, which can take a long time if the text file contains a lot of data.

Another limitation is the inability to store relations between different data. For example, a store may have both a list of customers and a list of orders—the orders come from customers. With a text file, there's no easy way to link an order in one list with a customer in another list.

A database does not have these limitations. Specific data may be quickly retrieved through keys and indexes, and different data may be easily linked.

This chapter used the Northwind Traders database. First, you learned how to obtain and install this database. After creating a new Windows application, you then created a connection between Visual Basic 2005 and the database. In doing so, you selected the database format, a provider suitable for that format, and the path to and the name of the database.

Next, you learned how to use Server Explorer, a tool provided by Visual Basic 2005 that enables you to view databases on your computer without having to open or create an application.

The code components used for database-access code are organized in the .NET library under the name ADO.NET. ADO was an acronym for ActiveX Data Objects, a Microsoft data-access technology that was the predecessor to ADO.NET.

As you learned in this chapter, accessing the Northwind Traders database involves the following steps:

1. Establish a connection to the database.
2. Define the commands you want to make to the database.
3. Define a data adapter that packages the database connection and commands.
4. Create a DataSet and then fill it using the data adapter.
5. Fill a control (for example, a DataGridView) from the DataSet.

You created an application that implemented these steps and filled a DataGridView control with data from four fields of the Customers table of the Northwind Traders database.

The project you created in this chapter is a Windows application. In the next chapter, you will learn to create a similar project that is a web application.

Quiz

1. What is a data provider?
2. What does Server Explorer enable you to do?
3. What is a table?
4. What may each column in a table also be called?
5. What may each row in a table also be called?
6. What is ADO.NET?
7. What class represents a connection to a data source?
8. What class would you use to execute commands to a database?
9. What class would you use to package a database connection with a set of data commands?
10. What is a DataSet?

Web Applications

Throughout this book we have been writing Windows applications. Indeed, many of the applications with which you interact are Windows applications. For me, it is a rare day that I don't work with Microsoft Word and Outlook, for example.

However, I am, and perhaps you are as well, interacting ever more frequently with web applications. One common type of web application is e-commerce, the *e* standing for electronic. For example, if you go to the website of Amazon or another online bookseller, you select a book (hint: this one) or another product, put the selected product in a virtual shopping cart, when finished go to a virtual check-out line, enter your credit card information (which better not be virtual), and make a purchase. You then can go to the website of the overnight delivery service and track the shipment as it wends its way across the country (or world) to you.

In this chapter, you will learn how to create a web application that displays information from a database, similar to the Windows application you created in Chapter 14.

ASP.NET

ASP.NET is a term you likely will hear of soon after you start creating web applications. ASP.NET refers to the code components used for web applications, similar to how ADO.NET refers to the code components used for database access.

As with ADO.NET, you already know the “.NET” portion of ASP.NET. ASP is an acronym for Active Service Pages, a Microsoft web application technology that preceded ASP.NET. For those of you who are familiar with ASP, ASP.NET is much easier to work with. ASP intermixed HTML with script code. By contrast, ASP .NET enables you to develop web applications in almost the same manner as Windows applications.

ASP.NET started with Visual Studio 2005’s predecessor, Visual Studio.NET. The version number of ASP.NET then was 1.x (first 1.0, then 1.1). With Visual Studio 2005, the version number is 2.x, starting with 2.0.

There are other, competing technologies for the creation of web applications. ASP.NET is Microsoft’s, and consequently the one heavily supported in Visual Studio 2005.

Internet Information Services

Visual Studio 2005 requires one of the following operating systems: Windows 2000 Professional, Windows XP Home or Professional, Windows 2000 Server, or Windows 2003 Server. On all but Windows XP Home Edition, Internet Information Services (IIS) is an optional component that may be installed with the operating system. IIS may not actually be installed on your computer because it may not be part of the default installation of your operating system. However, if IIS is not installed, you can add it as described in this section.

NOTE *You cannot install IIS on Windows XP Home Edition unless you make some Registry changes that are not supported by Microsoft and therefore probably are not a good idea to try.*

Unlike ASP.NET 1.x and Visual Studio.NET, ASP.NET 2.x and Visual Studio 2005 do not require you to install IIS to create web applications that run locally (that is, on your computer). Nevertheless, unless you have Windows XP Home Edition, installing IIS does give you more options, such as making your web pages accessible from more than your local computer, and it costs you nothing.

Determining If IIS Is Already Installed

To determine if IIS is already installed on your computer, open Add/Remove Programs from the Control Panel. From the left menu bar, choose Add/Remove Windows Components. This will display the Windows Components Wizard, shown in Figure 15-1.

In Figure 15-1, Internet Information Services (IIS) is checked, but with a dark background. This indicates some but not all of the components of IIS are installed. If IIS is checked but with a white background, as is Internet Explorer in Figure 15-1, then all of the components of IIS are installed. If IIS is unchecked, as is the Indexing Service in Figure 15-1, then IIS is not installed.

If IIS is checked, but with a dark background as in Figure 15-1, then you need to check which of its components are installed. To do so, in the Windows Components Wizard, highlight Internet Information Services (IIS) and click the Details button. This will display, as shown in Figure 15-2, a dialog box showing the individual components of Internet Information Services (IIS).

In Figure 15-2, almost all of the check boxes are checked because those components happen to be installed on my computer. This may not be the case on your computer, depending on which components of IIS you previously may have installed.

You don't need the FTP (File Transfer Protocol) and SMTP (Simple Mail Transfer Protocol) services, but I recommend you install the other components.

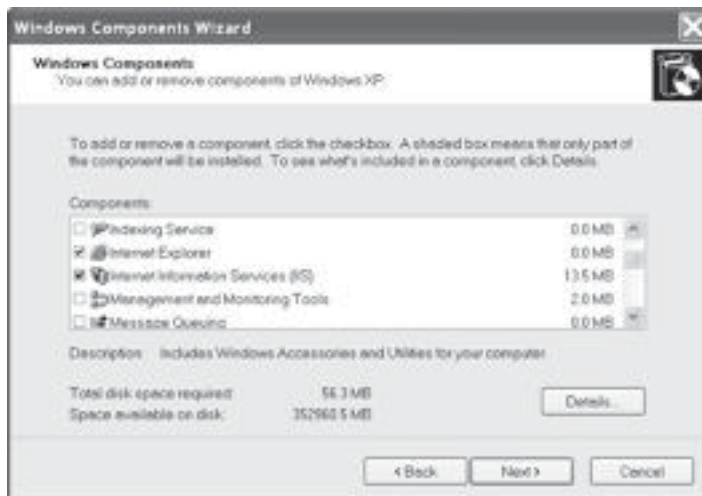


Figure 15-1 Windows Components Wizard

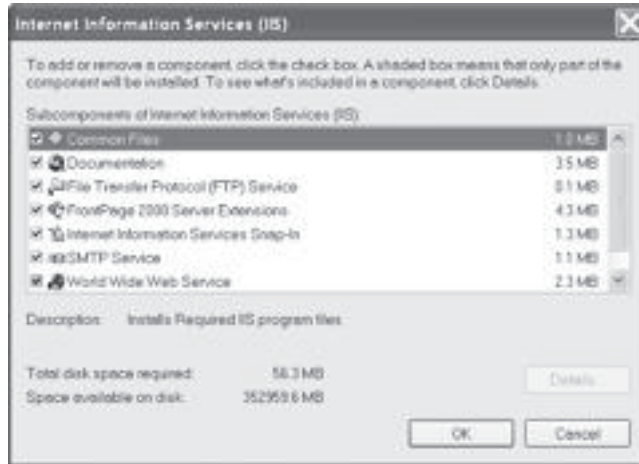


Figure 15-2 IIS components

Installing IIS

If you do need to install IIS or components of it, first locate the installation CD of your operating system, because you may need it. If IIS is unchecked in the Windows Components Wizard (see Figure 15-1), first check it and then click the Next button. If IIS is checked in the Windows Components Wizard but the check box has a dark background, just click the Next button.

Clicking the Next button displays the Internet Information Services (IIS) dialog box shown in Figure 15-2. Choose all of the components by checking the boxes that are not already checked, again with the possible exception of the FTP and SMTP services. Then click the OK button, which will return you to the Windows Components Wizard. In the Windows Components Wizard, after verifying that you have your operating system installation CD in your CD-ROM drive, click the Next button and continue to proceed until you are finished adding the IIS components. If prompted to do so, restart your computer.

Start the IIS Admin Service

The IIS Admin Service is, as its name suggests, a service used to administer IIS. Although there are alternative methods of administering IIS, using the IIS Admin Service may be the easiest.

Open the Administrative Tools folder in Control Panel. This folder is shown in Figure 15-3.

Next, choose the Services shortcut to open the Services folder. Choose the Extended tab and highlight IIS Admin. As Figure 15-4 shows, to the left is a description of the IIS Admin service as well as options to stop, pause, and restart the service.

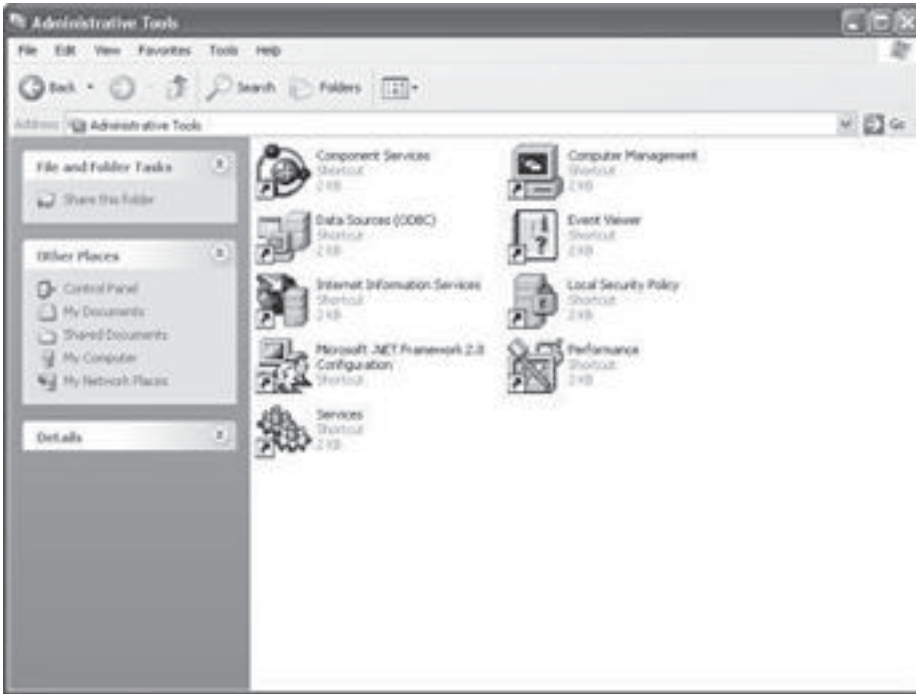


Figure 15-3 Administrative Tools folder in Control Panel

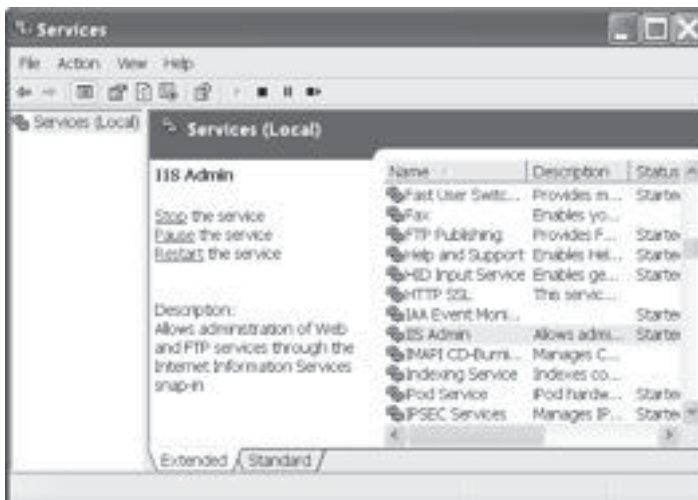


Figure 15-4 Services folder with IIS Admin service selected

The options are to stop, pause, and restart the service because the service already is started. In that event, you have confirmed that the IIS Admin service has started, and you are done with this step.

However, if the IIS Admin service had stopped or never started, the option instead would be to start the service, as shown in Figure 15-5. In that event, you would choose Start to start the service.

Starting the Default Website

Once you have confirmed that the IIS Admin service has started, close the Services folder and go back to the Administrative Tools folder shown in Figure 15-3. Next, choose the Internet Information Services shortcut to open the Internet Information Services dialog box shown in Figure 15-6.

Click the + sign next to the local computer name (mine is JAKXP; yours is likely different) and then click the + sign next to the Web Sites folder below it. Figure 15-7 shows a subfolder named Default Web Site.

If Default Web Site is followed by a parenthetical indicating it is stopped, right-click Default Web Site and choose Start from the shortcut menu.

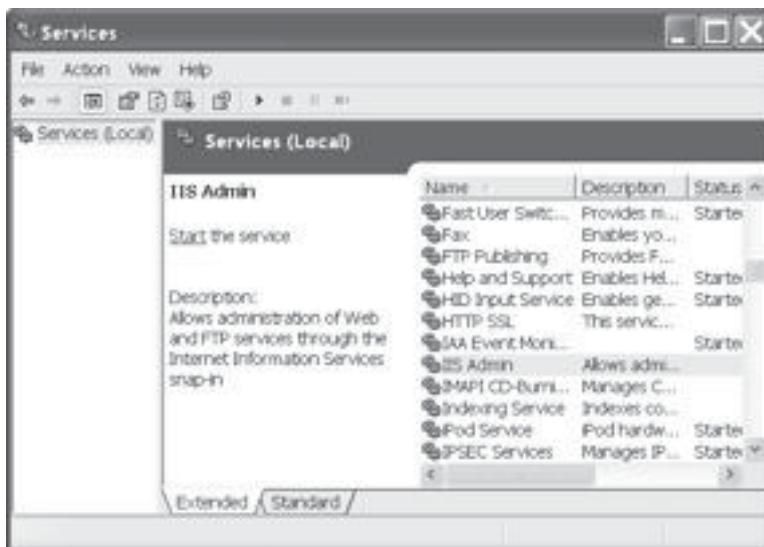


Figure 15-5 Option to start the IIS Admin service

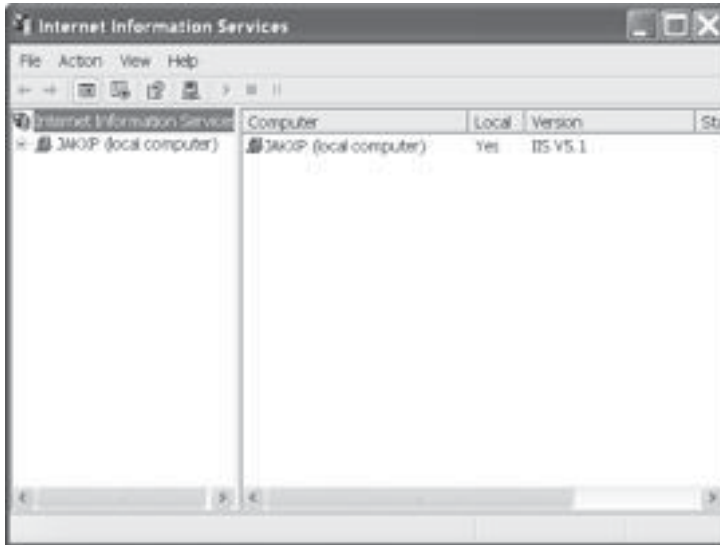


Figure 15-6 Internet Information Services dialog box

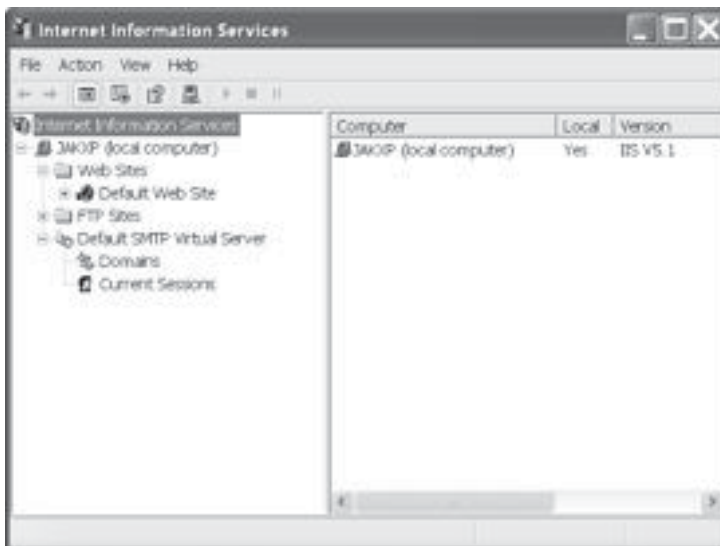


Figure 15-7 Default Web Site in the Internet Information Services dialog box

URL

Your home or apartment has an address by which it may be located. A web page similarly has an address by which you may locate it through your web browser.

The address of your home or apartment usually is in the form of a number followed by a street name, such as 1313 Mockingbird Lane. The address of a web page, referred to as a URL (an acronym for Uniform Resource Locator), similarly has a certain form.

The following explanation will use as an example the URL for Microsoft's home page, <http://www.microsoft.com/default.aspx>.

The first part of the address (here, <http://>) indicates what protocol to use. HTTP is an acronym for Hypertext Transfer Protocol. HTTP defines how messages are formatted and transmitted, and what actions web servers and browsers should take in response to various commands. For example, when you enter a URL in your browser, this actually sends an HTTP command to the web server directing it to fetch and transmit the requested web page.

There are protocols other than HTTP. One is similarly named HTTPS, a secure form of HTTP often used for credit card transactions on the Internet. Another is FTP, File Transfer Protocol, used for transferring files.

The second part of the address (here, www.microsoft.com) is the domain name where the resource is located. Domain names commonly start with www, short for World Wide Web, and end with com (for commercial) or another extension, such as net or org. In between is a name (here, Microsoft), which often corresponds to the organization or individual who owns the website. For example, my website is <http://www.genghiskhent.com>, based on my students' fond (?) nickname for me, Genghis Khent.

The third part of the address is the specific web page being accessed (here, default.aspx). Web pages are named in a similar fashion to other files, a descriptive name followed by a dot and an extension.

In Windows applications, the extension indicates the application used to open the file, such as .doc for Microsoft Word, .xls for Microsoft Excel, and so forth. Web pages may have extensions such as .htm and .html. The .aspx extension indicates that the web page is part of an ASP.NET application.

Your Computer as the Web Server

A web server is a computer that delivers (serves up) web pages. For example, if you visit Microsoft's home page, <http://www.microsoft.com/default.aspx>, by entering that address in your web browser (such as Internet Explorer, Netscape, or Mozilla),

a computer somewhere on the Internet fetches a page on the Microsoft website and sends its content to your browser, where that content then is displayed in your computer's web browser.

In this chapter, however, your computer will act as the web server for the web applications you will be creating.

Type the URL **http://localhost/** in your web browser (this won't work if you have Windows XP Home, as already mentioned). Figure 15-8 shows the web page that then displays on the Windows XP operating system.

You may legitimately wonder, what is localhost? You have heard of microsoft.com and other .com and .net URLs, but localhost may be a new one for you. The answer is localhost is your computer, which now is acting as a web server.

Virtual and Physical Paths

When you type `http://www.microsoft.com` in your web browser, you are accessing a page stored on the hard drive of a computer Microsoft is using as a web server.



Figure 15-8 Default web page

Similarly, when you typed `http://localhost` and the web page shown in Figure 15-8 was displayed, that web page also was stored on your computer's hard drive.

By default, `http://localhost` maps to the `C:\inetpub\wwwroot` folder on your hard drive. You can confirm this by right-clicking Default Web Site (refer to Figure 15-7) and choosing Properties from the shortcut menu to display the Default Web Site Properties dialog box, which is shown in Figure 15-9 with the Home Directory tab chosen. The local path is `c:\inetpub\wwwroot`.

The address bar in Figure 15-8 shows that the URL of the web page is `http://localhost/localstart.asp`. Thus, the URL `http://localhost/localstart.asp` maps to the file `C:\inetpub\wwwroot\localstart.asp` on your hard drive.

The web URL `http://localhost/localstart.asp` is known as the *virtual path* to the web page. The file path `C:\inetpub\wwwroot\localstart.asp` is known as the *physical path* to the web page. However, they both point to the same place.

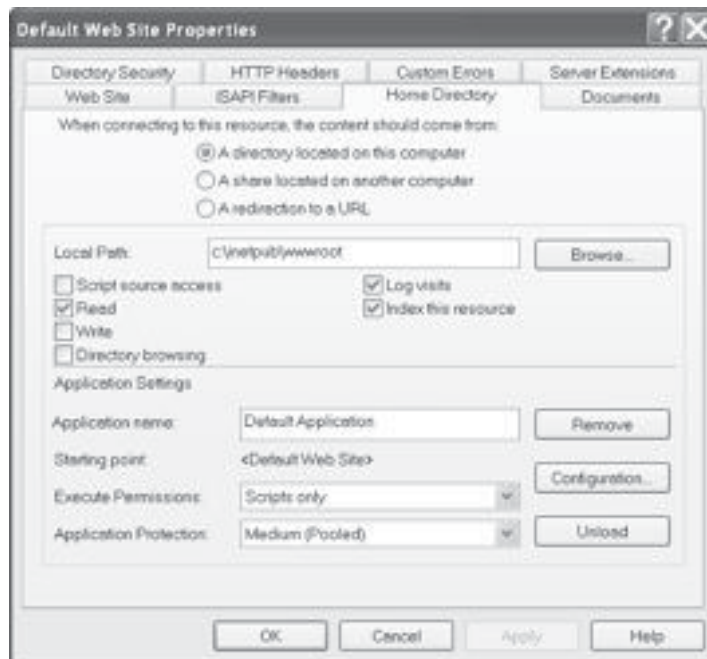


Figure 15-9 Default Web Site Properties dialog box

Creating a Web Application

Creating a web application is different from creating a Windows application. You use the File | New | Website menu command instead of the File | New | Project menu command.

The File | New | Website menu command displays the New Web Site dialog box shown in Figure 15-10.

The top pane shows available templates. Choose ASP.NET Web Site. This is the proper choice for creating a website with ASP.NET support, which is what we want to do here.

In the Location drop-down box, choose File System. The other choices, FTP and HTTP, both protocols discussed earlier in this chapter, are for creating ASP.NET websites on other computers. In this chapter, you will be creating the website on your computer.

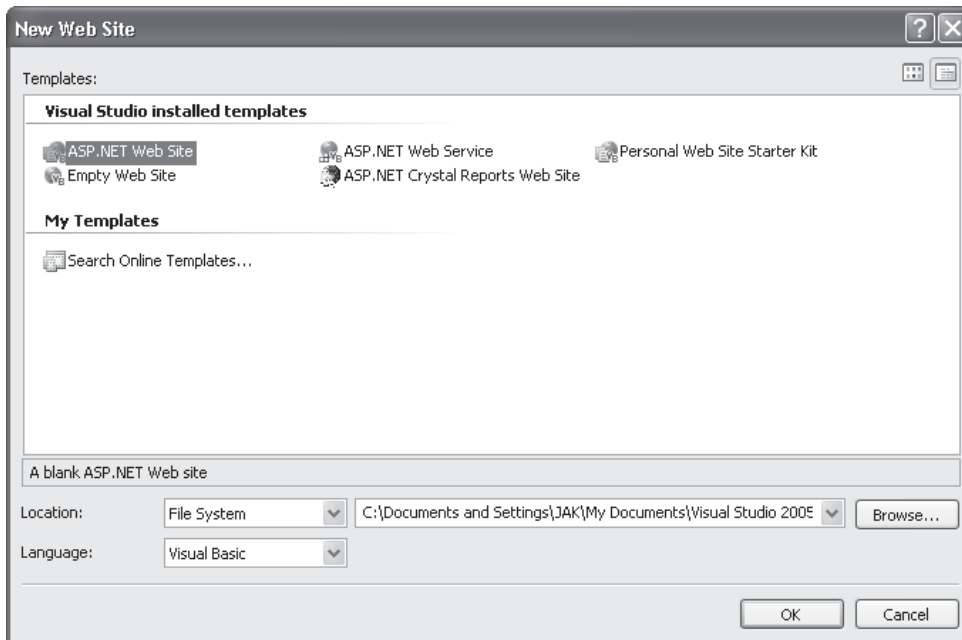


Figure 15-10 New Web Site dialog box

In the Language drop-down box, choose Visual Basic. The other choices, Visual C# and Visual J#, are other languages in Visual Studio 2005 that you may use to create an ASP.NET application.

Click the Browse button to select where on your hard drive you wish to create the files for the ASP.NET web application. I chose a Visual Basic folder I previously had created in the Visual Studio Projects folder under My Documents. After the path to the Visual Basic folder (for example, D:\Documents and Settings\JAK\My Documents\Visual Studio Projects\Visual Basic\) I typed WebSite1 for the name of the project. Of course, you could choose a different location or name for your project.

When finished, click the OK button, and Visual Studio 2005 will create a barebones but working ASP.NET application.

ASP.NET Development Server

When Visual Studio 2005 is finished creating the ASP.NET application, run the application by choosing Start or Start Without Debugging from the Debug menu. The result will be a blank web page, as shown in Figure 15-11.

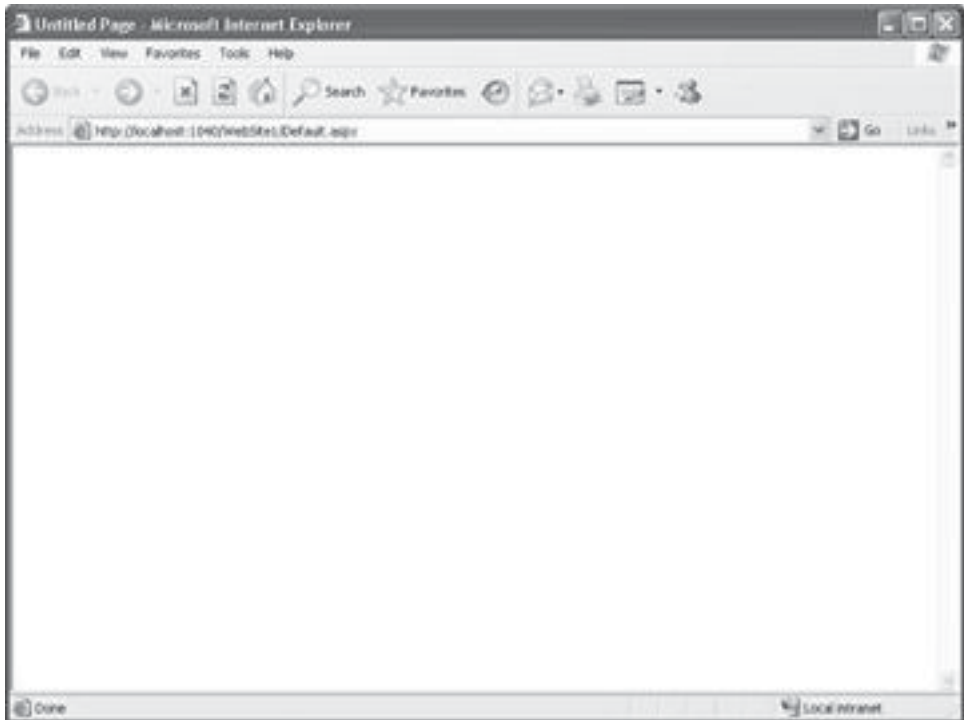


Figure 15-11 ASP.NET web page

The URL shown in the address bar of the web browser in Figure 15-11 is `http://localhost:1040/WebSite1/Default.aspx`. The `http://localhost` part of the URL is explained in the earlier section “Your Computer as the Web Server.” `WebSite1` is the name of the web application, and `default.aspx` is the name of the web page (or web form) that Visual Studio 2005 creates by default, much like a Windows form is created by default when you create a Windows application.

What is new, and its meaning may not be immediately clear, is the “:1040” following `localhost`. The colon (:) means that the number following is a port number (here, 1040).

NOTE *The particular port number assigned by Visual Studio 2005 may be different from 1040.*

A port is a logical (as opposed to physical) connection in a computer. For example, when you access a web page with your web browser, your request (and the web server’s response) goes through port 80.

As mentioned in the earlier section “Internet Information Services,” ASP.NET 2.x and Visual Studio 2005 do not require you to install IIS to create web applications that run locally (that is, on your computer as opposed to a computer elsewhere on the Internet). Instead, local web applications are handled through the ASP.NET Development Server, which uses various port numbers (here, 1040).

You may have an icon for the ASP.NET Development Server in your system tray. If so, double-click it. The ASP.NET Development Server dialog box will appear, as shown in Figure 15-12.

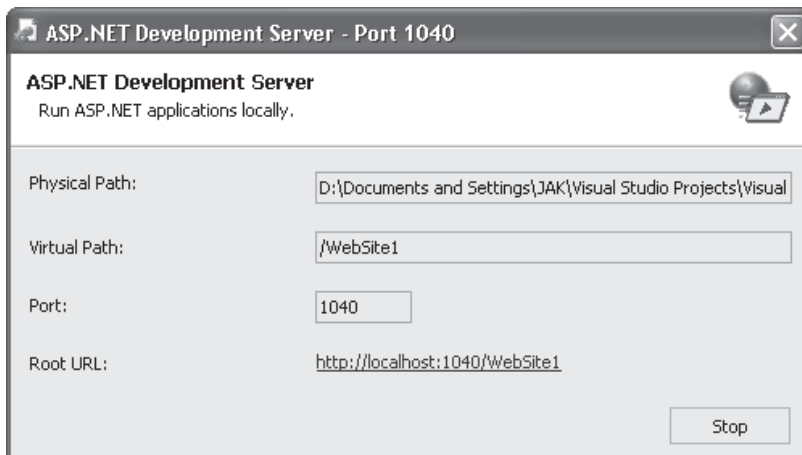


Figure 15-12 ASP.NET Development Server dialog box

The ASP.NET Development Server dialog box shows the following information (though not in this order from top down):

- **Physical Path** The location you chose in the New Web Site dialog box shown in Figure 15-10
- **Port** The port chosen by the ASP.NET Development Server for access to local web applications (here, 1040)
- **Root URL** The root or base for web applications (`http://localhost:1040`), followed by the name of this web application (here, `WebSite1`)
- **Virtual Path** The path from the root URL of `http://localhost:1040` to your web application

That is about all we can do for now with this blank web application. Close the ASP.NET Development Server dialog box shown in Figure 15-12 and the blank web page shown in Figure 15-11.

ASP.NET Application IDE

Figure 15-13 shows the Integrated Development Environment (IDE) for the ASP .NET application we created by clicking OK in the New Web Site dialog box shown in Figure 15-10.

As with Windows applications, the form in web applications, often called a *web form*, also has both a design view (shown in Figure 15-13), complete with a Toolbox and Solution Explorer, and a code view, shown in Figure 15-14.

This similarity between the IDEs for Windows and web applications makes it easier for you to learn to develop web applications.

Although the respective IDEs of Windows and web applications are similar, they are not the same. For example, the web form has, in addition to design and code views, an HTML view, shown in Figure 15-15 and accessed by clicking the Source tab, in which you can view the HTML code of the form, which after all is a web page.

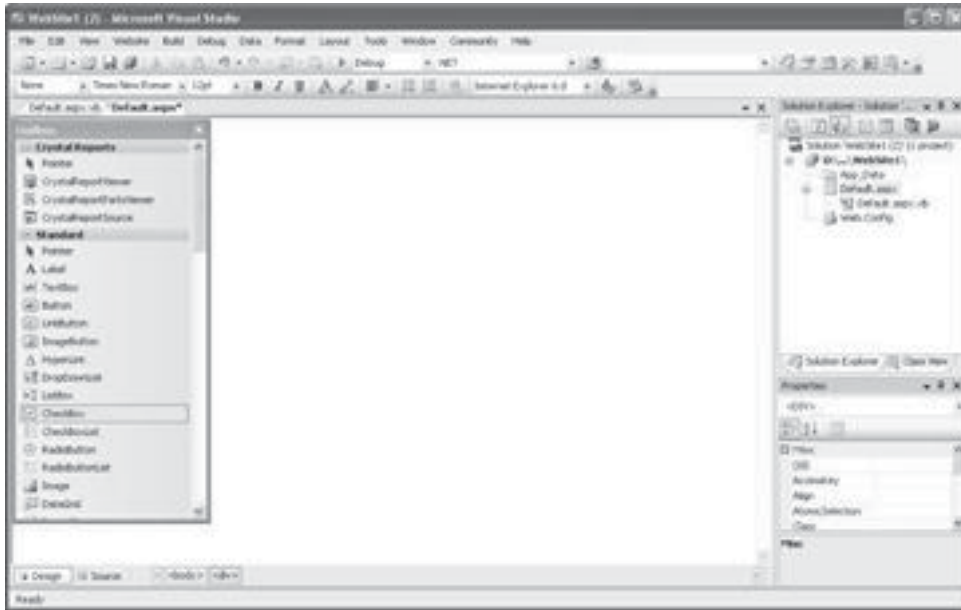


Figure 15-13 ASP.NET application IDE

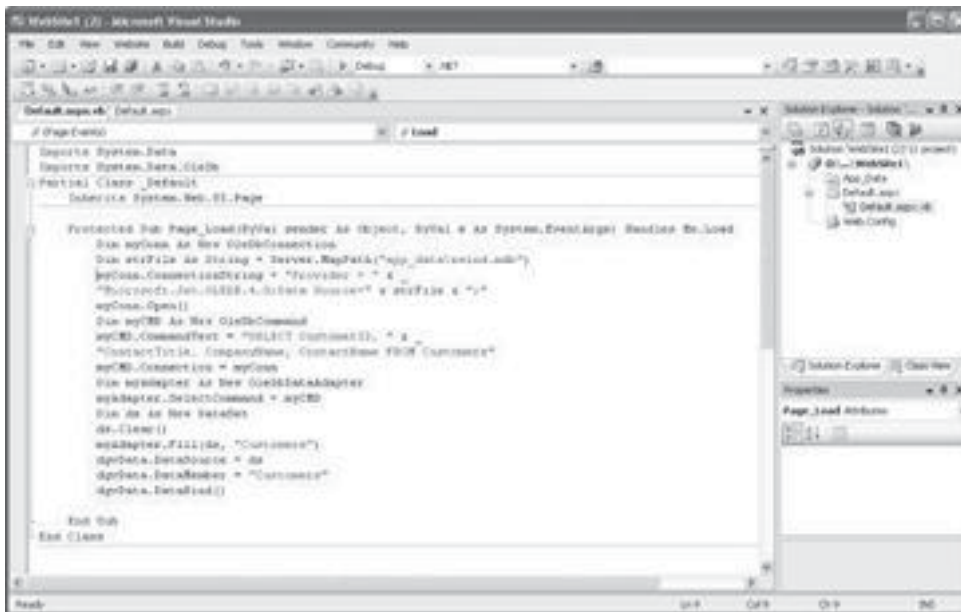


Figure 15-14 Code view

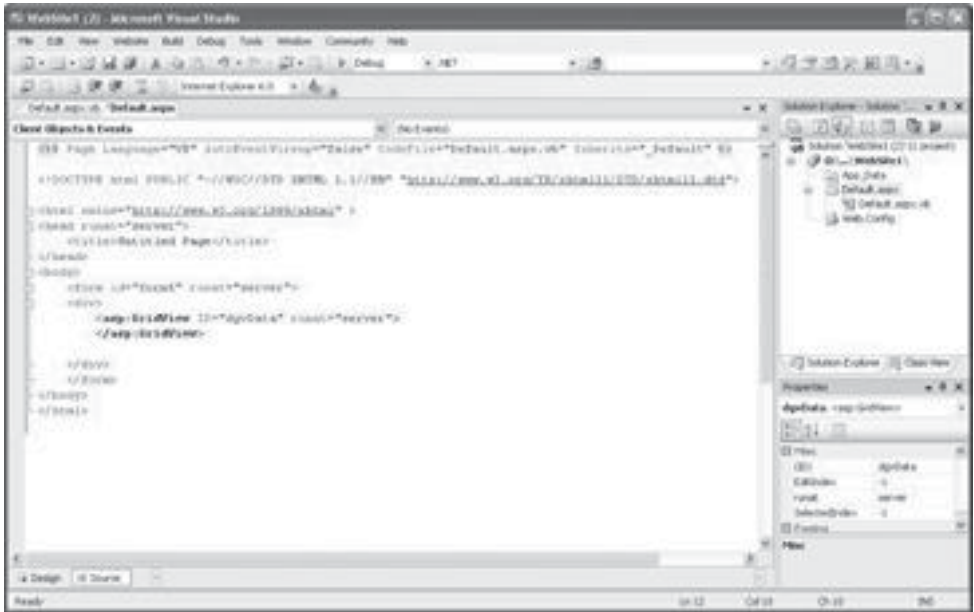


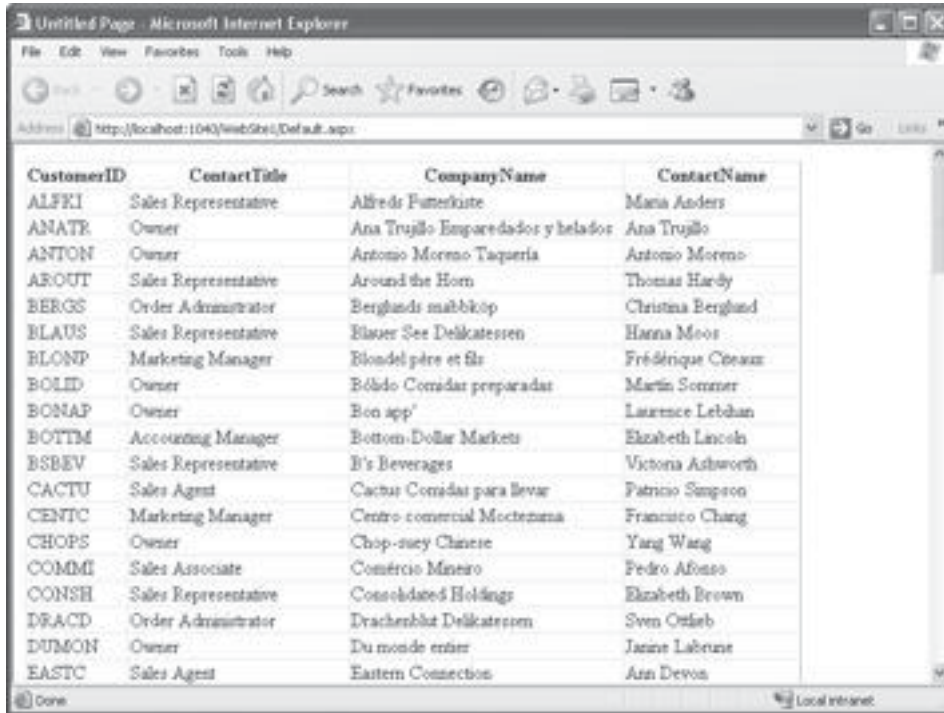
Figure 15-15 HTML view of the form

Creating a Database Web Application

We will now create a web application that parallels the Windows application we created in Chapter 14. That Windows application displayed in a `DataGridView` control the contents of four fields of the Customers table of the Northwind Traders database. The web application you will create similarly will display the contents of the same four fields of the Customers table of the Northwind Traders database, but in a web browser, as shown in Figure 15-16.

Adding a GridView Control

The Windows application we created in Chapter 14 has a `DataGridView` control through which we viewed the database information. For whatever reason, the web application equivalent of the Windows `DataGridView` control does not have the same name, but a slightly different one, `GridView`.



The screenshot shows a Microsoft Internet Explorer browser window displaying a web application. The address bar shows the URL `http://localhost:1040/webSite1/Default.aspx`. The main content area displays a table with four columns: CustomerID, ContactTitle, CompanyName, and ContactName. The table contains 20 rows of data, including entries like ALFKI, ANATR, ANTON, AROUT, BERGS, BLAUS, BLONP, BOLID, BONAP, BOTTM, BSBEV, CACTU, CENTC, CHOPS, COMMI, CONSH, DRACD, DUMON, and EASTC.

CustomerID	ContactTitle	CompanyName	ContactName
ALFKI	Sales Representative	Alfreds Futterkiste	Maria Anders
ANATR	Owner	Ana Trujillo Emparedados y helados	Ana Trujillo
ANTON	Owner	Antonio Moreno Taqueria	Antonio Moreno
AROUT	Sales Representative	Around the Horn	Thomas Hardy
BERGS	Order Administrator	Berglunds snabbköp	Christina Berglund
BLAUS	Sales Representative	Blauser See Delikatessen	Hanna Moos
BLONP	Marketing Manager	Blondel père et fils	Frédérique Citeaux
BOLID	Owner	Bólido Comidas preparadas	Martin Sommer
BONAP	Owner	Bon app'	Laurence Letahan
BOTTM	Accounting Manager	Bottom-Dollar Markets	Elizabeth Lincoln
BSBEV	Sales Representative	B's Beverages	Victoria Ashworth
CACTU	Sales Agent	Cactus Comidas para llevar	Patricio Simpson
CENTC	Marketing Manager	Centro comercial Moctezuma	Francisco Chang
CHOPS	Owner	Chop-suey Chinese	Yang Wang
COMMI	Sales Associate	Comércio Mineiro	Pedro Afonso
CONSH	Sales Representative	Consolidated Holdings	Elizabeth Brown
DRACD	Order Administrator	Drachenhut Delikatessen	Sven Ottlieb
DUMON	Owner	Du monde entier	Jeanne Labrousse
EASTC	Sales Agent	Eastern Connection	Ann Devon

Figure 15-16 Web application in action

Start with the web application you created in the previous section. View the web form in designer view and click the Design tab. Then look in the Toolbox for a GridView in the Data group, as shown in Figure 15-17.

If you don't see the GridView in the Toolbox, you need to add it. Right-click the Toolbox and select Choose Items... from the shortcut menu. This will display the Choose ToolBox Items dialog box shown in Figure 15-18.

Select the check box for the GridView whose namespace is System.Web.UI.WebControls. Next, click the OK button to close the Choose ToolBox Items dialog box. GridView should now be added to the Toolbox, as in Figure 15-17.

Once the GridView is in the Toolbox, you add it to the web form by dragging and dropping or double-clicking, just as you would add a control to a Windows form.

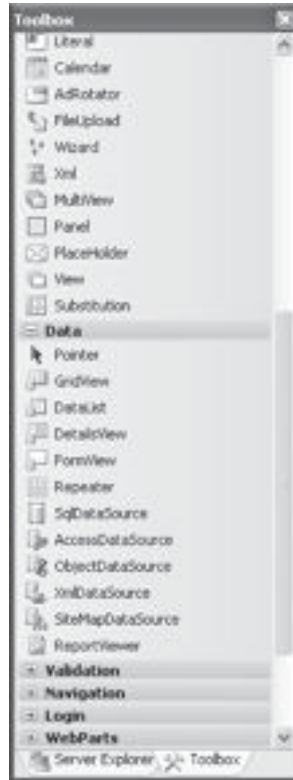


Figure 15-17 GridView in Toolbox

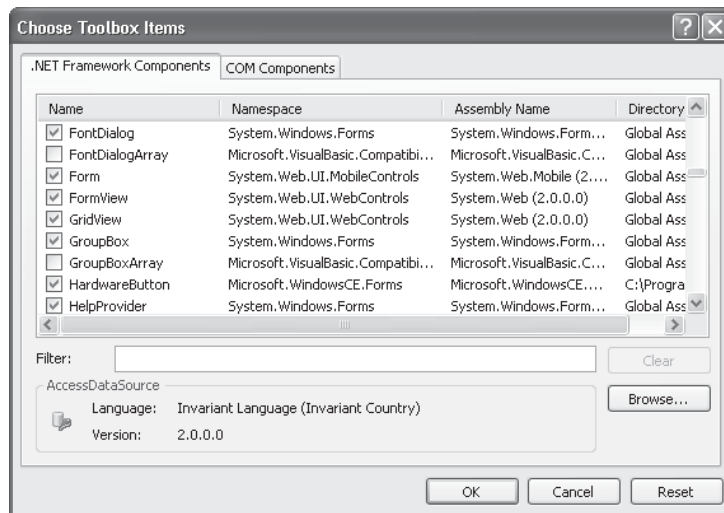


Figure 15-18 Choose ToolBox Items dialog box

When you add the GridView control, a GridView Tasks pane displays, as shown in Figure 15-19. You may accept the default values in this pane. However, using the Properties window, rename the GridView control (using its ID property) `dgvData` to keep its name consistent with the DataGridView control in the Windows application, because we are attempting to port the code from the Windows application to this web application.

Locating the Database on the Web Server

The GridView is the only control we will be adding to the web form. There is no web application equivalent of the OpenFileDialog control that we used in the Windows application in Chapter 14.

Additionally, we would not want the user to select the location of the database. In a Windows application, the database often may be on the user's computer. Therefore, it is logical to have the user locate and select the database file using the OpenFileDialog control. By contrast, in a web application, the database will not be on the user's computer, but rather a web server elsewhere on the Internet. For security reasons, the user should not be permitted to browse the files on the web server as the user would for the files on their own computer. Instead, the web application should specify where the database file is.

Often the database is located in a subfolder of the web application to ease the task of locating it through code, as will be discussed next. By default, the ASP.NET application created by Visual Studio 2005 has a subfolder named `App_Data`, likely short for application data. Copy the `nwind.mdb` (or `Northwind.mdb`) file into the `App_Data` folder from wherever you saved `nwind.mdb` when creating the Windows database application in Chapter 14.

Now that you have located the database on your hard drive within the web application files, the remaining task is how to locate the database in code.

As discussed in Chapter 14, the `ConnectionString` property of the `OleDbConnection` object requires the path to and name of the database file. In the Windows database application in Chapter 14, you obtained the path to and the

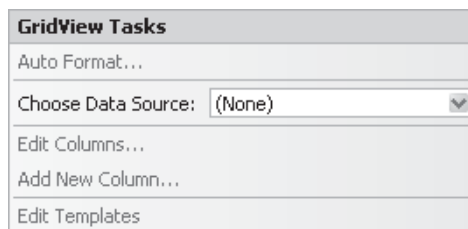


Figure 15-19 GridView Tasks pane

name of the database file (represented by the String variable `strFile`) by the `FileName` property of the `OpenFileDialog` control:

```
Dim myConn As New OleDbConnection
Dim dr As DialogResult
dr = dlgOpen.ShowDialog()
If dr = DialogResult.OK Then
    Dim strFile As String = dlgOpen.FileName
    myConn.ConnectionString = "Provider = " & _
        "Microsoft.Jet.OLEDB.4.0;Data Source=" & strFile & ";"
```

You cannot obtain the path to and the name of the database file the same way in this web application because there is no `OpenFileDialog` control. However, you know where the database file is located, in the `app_data` subfolder of the web application. Thus, the virtual path to the database is `http://localhost:1040/WebSite1/app_data/nwind.mdb`.

However, the `ConnectionString` property requires the physical path, not the virtual path. In this case, you know the physical path because the database file is on your computer. However, when you are working with remote web servers, you may not always know the physical path, or even if you did, the administrator of that web server may change it. Therefore, you need to be able to translate the virtual path into a physical path.

The `HttpServerUtility` class, which also can be referred to as the `Server` class, has a `MapPath` method that returns the physical file path that corresponds to (is mapped to) the specified virtual path on the web server. The following statement assigns to the String variable `strFile` the physical path to the database file:

```
Dim strFile As String = Server.MapPath _
    ("app_data\nwind.mdb")
```

NOTE You may need to change the reference to `nwind.mdb` to `Northwind.mdb` if the latter is the file name on your computer.

The `MapPath` method starts by mapping the physical path that corresponds with the virtual path to the web application, `http://localhost:1040/WebSite1`. The argument then is appended to that physical path. The method then returns the physical path that corresponds with the full virtual path to the database file, `http://localhost:1040/WebSite1/app_data/nwind.mdb`.

Accordingly, the preceding code from Chapter 14 would be replaced with the following:

```
Dim myConn As New OleDbConnection
Dim strFile As String = Server.MapPath _
```

```
("app_data\nwind.mdb")
myConn.ConnectionString = "Provider = " & _
"Microsoft.Jet.OLEDB.4.0;Data Source=" & strFile & ";"
```

Adding Code

The next step is to write code. To do so, go to the code view of the web form.

First, we will import the `System.Data` and `System.Data.OleDb` namespaces for the same reason we did in Chapter 14. The two `Imports` statements go immediately above the class name:

```
Imports System.Data
Imports System.Data.OleDb
Partial Class _default
```

Second, as in Chapter 14, all the code will go in the `Load` event, this time of the web page. This event procedure belongs to the `Page` object, which represents the web form.

To create an event procedure, similar to Windows forms, you choose (Page Events) from the left drop-down box. Then you choose the event (here, `Load`) from the right drop-down box. This creates an event procedure stub. Then you write code so your `Page_Load` event procedure reads as follows:

```
Private Sub Page_Load (ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles Me.Load
    Dim myConn As New OleDbConnection
    Dim strFile As String = Server.MapPath _
        ("app_data\nwind.mdb")
    myConn.ConnectionString = "Provider = " & _
        "Microsoft.Jet.OLEDB.4.0;Data Source=" & strFile & ";"
    myConn.Open()
    Dim myCMD As New OleDbCommand
    myCMD.CommandText = "SELECT CustomerID, " & _
        "ContactTitle, CompanyName, ContactName FROM Customers"
    myCMD.Connection = myConn
    Dim myAdapter As New OleDbDataAdapter
    myAdapter.SelectCommand = myCMD
    Dim ds As New DataSet
    ds.Clear()
    myAdapter.Fill(ds, "Customers")
    dgvData.DataSource = ds
    dgvData.DataMember = "Customers"
    dgvData.DataBind()
End Sub
```

This code differs in only two substantive respects from the corresponding code in the Form Load event procedure in Chapter 14. First is the use of the `MapPath` method, as discussed in the earlier section “Locating the Database on the Web Server.” The second is the last statement, the call to the `DataBind` method of the `GridView`. This method is commonly used in web applications to bind data from a source (here, a `DataSet`) to a control (here, a `GridView`).

CAUTION *If you don't call the `DataBind` method, the web application will run without error, but the `GridView` will be blank, because it was not bound to the data source.*

Run your web application from the Debug menu, again just as you would a Windows application. The web page should display, with the `GridView` filled with information, as shown previously in Figure 15-16. When you are done, close the web page using its close button to close the application.

Conclusion

Of course, there is much more to web applications. Entire courses and books are devoted to web applications. However, this chapter should give you an overview of how to create a working web application that displays information from a database.

This is the last chapter in this book. However, it should not be the last chapter in your learning Visual Basic 2005. Rather, this book hopefully has given you a good foundation for learning more.

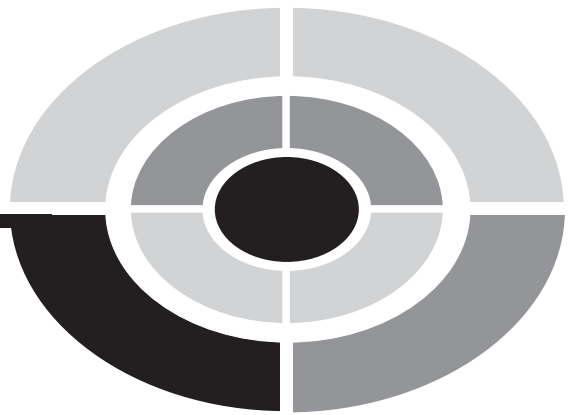
Quiz

1. What is ASP.NET?
2. What is a URL?
3. What is HTTP?
4. What does the `.aspx` extension indicate?
5. What is the difference between a virtual and a physical path to a web page?
6. What project template could you use to create a web application?



7. What is the web control that corresponds to the DataGridView control used in Windows applications?
8. What is the method of the HttpServerUtility class that returns the physical file path that corresponds to (is mapped to) the specified virtual path on a web server?
9. What is the name of the class that is the web application equivalent of the Form class in a Windows application?
10. What is the method of the GridView that needs to be called in a web application so the GridView will not be blank?

This page intentionally left blank



Final Exam

1. What is an IDE?
2. What is a computer program?
3. What is a programming language?
4. What is machine language?
5. What does “higher level” mean in the context of a programming language?
6. What does “lower level” mean in the context of a programming language?
7. What is the purpose of a compiler?
8. What is a class in a programming language?
9. What is an object of a class?
10. What are namespaces used for?
11. What is a property of a class?
12. What are characteristics of a Windows application?

13. What is an event of a class?
14. What is an event procedure?
15. What is the purpose of the assignment operator?
16. What is the purpose of the Toolbox?
17. How do you add a control from the Toolbox onto your form?
18. What is the purpose of the Name property of a control?
19. What is a naming convention?
20. What are purposes of the text displayed by a Label control?
21. What is a parameter of an event procedure?
22. What does a data type signify?
23. What is the purpose of a variable?
24. Does Visual Basic 2005 by default require you to declare a variable before you refer to it in code?
25. What is a local variable?
26. Do you have to assign a value to a variable when you declare it?
27. What is a difference between a constant and a variable?
28. Do you have to assign a value to a constant when you declare it?
29. What is the significance of operator precedence?
30. Which operator provides only the remainder resulting from division?
31. Which operator has precedence, an arithmetic operator or the assignment operator?
32. What is the purpose of the Parse method of the Integer class?
33. What is the purpose of the ToString method of the Integer class?
34. What is a method of a class?
35. What does the WriteLine method of the Debug class do?
36. What is the data type of the result of a comparison performed by a comparison operator?
37. Which operators have precedence, comparison or arithmetic?
38. What is the purpose of a logical operator?
39. Which logical operator operates on only one operand rather than two?

40. Which operators have precedence, comparison or logical?
41. What does modal mean?
42. What is the return value of the InputBox function if the OK button is clicked?
43. What is the return value of the InputBox function if the Cancel button is clicked?
44. What is an exception?
45. What does the TryParse method of the Integer class do?
46. Which two controls are commonly used with the If control structure?
47. What is the primary difference between the If...ElseIf statement and the Select Case control structure?
48. What is a loop?
49. What is a difference between the Do statement and the For...Next and While...End While statements?
50. What is a difference between the For Each...Next loop and the For...Next loop?
51. What is an array?
52. What is the difference between declaring an array variable and a scalar variable?
53. What is the lowest index of an array?
54. What is the relationship between the number of elements in an array and the highest index in that array?
55. What is a procedure?
56. What is the difference between a subroutine and a function?
57. What does the Private access specifier do when applied to a method?
58. What does calling a subroutine do?
59. What is the difference between the ByVal and ByRef attributes of a parameter?
60. What are some reasons for writing your own procedures?
61. Is a message box modal or modeless?
62. What value is returned by the Show method of the MessageBox class?

63. Do buttons in a message box automatically have a DialogResult value?
64. What is the data type of a variable you may use to store the return value of the Show method of the MessageBox class?
65. What method do you use to display a modal form?
66. What is the return value from showing a dialog form?
67. Do buttons in a dialog form you create automatically have a DialogResult value?
68. What method do you use to display a form as modeless rather than modal?
69. What class represents a main menu?
70. Is the Click event raised for all menu items?
71. How do you gray out a menu item so it is not available when it should not be?
72. What does the Items collection of the MenuStrip component contain?
73. What class represents the shortcut or context menu?
74. What does the Items collection of the ContextMenuStrip component contain?
75. What are the different alternatives for having a context menu item's functionality handled by the corresponding main menu item?
76. What class represents a toolbar?
77. What class represents each item on a toolbar?
78. What does the Items collection of the ToolStrip component contain?
79. What are advantages of a toolbar over a corresponding menu?
80. What are different alternatives for having a toolbar item's functionality handled by the corresponding main or context menu item?
81. What method do you use to show an Open dialog box?
82. What is the return value of showing an Open dialog box?
83. What is the property of the OpenFileDialog class whose value is the file chosen by the user in an Open dialog box?
84. What method of the SaveFileDialog class do you use to show a Save dialog box?
85. What is the return value of showing a Save dialog box?

86. What is the property of the SaveFileDialog class whose value is the name of the file to be saved?
87. What class may you use to read from a text file?
88. What class may you use to write to a text file?
89. What is a data provider?
90. What is a table?
91. What may each column in a table also be called?
92. What may each row in a table also be called?
93. What is ADO.NET?
94. What is a DataSet?
95. What is ASP.NET?
96. What is a URL?
97. What is HTTP?
98. What is the difference between a virtual and a physical path to a web page?
99. What is the method of the HttpServerUtility class that returns the physical file path that corresponds to (is mapped to) the specified virtual path on a web server?
100. What is the name of the class that is the web application equivalent of the Form class in a Windows application?

This page intentionally left blank



Answers to Quizzes and Final Exam

Chapter 1

1. Visual Studio 2005 includes, in addition to Visual Basic, support for other programming languages such as C++ and C#.
2. You need either the Windows 2003, XP, or 2000 operating system to install and run Visual Basic 2005.
3. You should use the Windows Application project template to start creating a Windows application.
4. IDE is an acronym for Integrated Development Environment. The term “development environment” refers to Visual Basic 2005’s role as an application to assist you in developing applications. The term “integrated” means the tools to design your application and write, test, and run your code are all together in one application.
5. A computer cannot do anything without step-by-step instructions from us telling it what to do. These instructions, written by a computer programmer, are called a computer program.
6. A programming language is used by computer programmers to write instructions for computers.
7. Machine language is a programming language that is understood by computers.
8. The term “higher level” means that a programming language such as Visual Basic 2005 is far closer to the structure and syntax of human language than to the ones and zeroes understood by a computer.
9. The term “lower level” means that a programming language such as machine language is far closer to the ones and zeroes understood by a computer than it is to the structure and syntax of human language.
10. In general, a compiler translates the code you write into corresponding machine language instructions. The compiler in Visual Basic 2005 translates the code into an intermediate language that then is translated into machine language.

Chapter 2

1. Designer view is the view of your form you would choose when you want to design your form, such as by resizing the form or adding controls to it.

- Code view is the view of your form you would choose when you want to view or write the code of your application.
- Programming languages, including Visual Basic, use classes to represent a person, place, thing, or concept.
- An object of a class is a single instance of a class, just like each of us could be said to be an object or instance of a Person class.
- Namespaces are used to organize code in a logical manner.
- A property is a characteristic or attribute of a class.
- A Windows application has a graphical user interface (GUI) and is event-driven.
- An event is something that happens to an object of a class, such as a result of user interaction.
- An event procedure contains code that executes when a specific event happens to a specific object.
- The purpose of the assignment operator is to assign the expression to its right to the variable or property to its left.

Chapter 3

- TextBox, Label, ListBox, and Button are examples of controls.
- The purpose of the Toolbox is to display controls that you can add to your form.
- You may add a control from the Toolbox on to your form either by double-clicking the control in the Toolbox or by dragging the control from the Toolbox and then dropping it on to the form,
- The Name property of a control is used to identify that control in code.
- A naming convention is a consistent method of naming, such as for naming controls.
- The value of the Text property of a Label control determines the text that will be displayed by the label.
- The text displayed by a label may identify another, adjacent control, or it may display data.
- The line-continuation character, an underscore (`_`) preceded by a space, is used to divide one, usually long line of code into multiple shorter lines of code.

9. A parameter represents information that is available to an event procedure.
10. The Handles clause indicates the object and the event of that object that is handled by an event procedure.

Chapter 4

1. A data type signifies whether the data is numeric, text, yes/no, and so forth.
2. A floating-point number is a number that may have a value to the right of the decimal point.
3. No, you cannot change the data type of a built-in property of a form.
4. The purpose of a variable is to store data of your choosing.
5. Yes, Visual Basic 2005, by the default setting of Option Explicit as On, requires you to declare a variable before you refer to it in code.
6. A local variable is a variable declared inside of a procedure.
7. A module-level variable is a variable declared outside of a procedure.
8. No, you do not have to assign a value to a variable when you declare it.
9. A constant's value cannot change during the life of the program, whereas a variable's value may change during the life of the program.
10. Yes, you have to assign a value to a constant when you declare it.

Chapter 5

1. The addition operator works with string as well as numeric variables.
2. Operator precedence determines, when there are two or more arithmetic operators, which arithmetic operation is done first.
3. You can override default operator precedence with parentheses.
4. The ^ operator raises a number to a specified power.
5. The / operator performs floating-point division, the remainder being preserved and expressed as a decimal, whereas the \ operator performs integer division, the remainder being dropped.
6. The Mod operator provides only the remainder resulting from division.
7. All arithmetic operators have precedence over the assignment operator.

8. The Parse method of the Integer class converts the string representation of an integer into actual integer values.
9. The ToString method of the Integer class converts an integer into its string representation.
10. A method is something an object of a class does.

Chapter 6

1. The WriteLine method of the Debug class outputs a line to the Output window.
2. The data type of the result of a comparison performed by a comparison operator is Boolean, True or False.
3. You can tell if the = operator is being used for assignment or comparison based on the code context in which the operator is used.
4. Yes, you can use comparison operators with strings as well as with numeric data types.
5. Option Compare determines whether a comparison is case sensitive or not.
6. The Like operator returns True or False depending on whether a string matches a specified pattern.
7. Arithmetic operators have higher precedence than comparison operators.
8. A logical operator is used to combine multiple comparisons.
9. Not is the logical operator that operates on only one operand rather than two.
10. Comparison operators have higher precedence than logical operators.

Chapter 7

1. Modal means a form must be closed before the application user can return to any other form in the application.
2. Modeless.
3. The return value of the InputBox function if the OK button is clicked is whatever the user typed in the input box.

4. The return value of the InputBox function if the Cancel button is clicked is an empty string.
5. If...Then, If...Then...Else, and If...ElseIf.
6. An exception is a problem that occurs while the program is executing that must be dealt with before the program can proceed.
7. The TryParse method of the Integer class converts the string representation of an integer into an integer, but also returns a Boolean value (True or False) indicating whether the conversion was successful.
8. The CheckBox and RadioButton controls.
9. The primary difference is that, in the If...ElseIf statement, the If and ElseIf clauses each may evaluate completely different expressions, whereas a Select Case control structure may evaluate only one expression, which then must be used for every comparison.
10. The Case Else part of a Select Case control structure performs the same purpose as an Else clause in an If control structure.

Chapter 8

1. A loop is a structure that repeats the execution of code until a condition becomes False.
2. A difference between a While...End While statement and a For...Next statement is that a For...Next statement generally is intended to run a fixed number of times, whereas a While...End While statement may run an indefinite number of times.
3. A difference between the Do statement and the For...Next and While...End While statements is that a Do statement may test a condition at the bottom as well as the top of the statement, whereas the For...Next and While...End While statements may test a condition only at the top of the statement.
4. The For Each...Next loop executes the statement block for each element in a collection, instead of a specified number of times as does the For...Next loop.
5. Examples of nesting are a loop within a loop, and an If control structure within a loop.
6. An array variable permits you to use a single variable to store multiple values, whereas a scalar variable may only store one variable at a time.

7. With an array variable, unlike a scalar variable, the array name is followed by a pair of parentheses, and within the parentheses you indicate the highest index of the array.
8. The lowest index of an array is zero.
9. The number of elements in an array is one greater than the highest index in that array because the index of the first element is zero.
10. Yes, if you declare an array without assigning a value to its elements, its elements have a default value, the value depending on the data type of the array.

Chapter 9

1. A procedure is a block of one or more code statements that execute when called upon to do so.
2. A subroutine does not return a value, whereas a function does.
3. An event procedure is a subroutine.
4. The Private access specifier limits access to the class in which the procedure was declared.
5. There is no difference between the Return and Exit Sub statements in subroutines; both end execution of the subroutine.
6. Calling a subroutine causes it to execute.
7. When a parameter has the ByVal attribute, any change to the value of the parameter in the called procedure does not affect the value of the corresponding argument in the calling procedure. By contrast, when a parameter has the ByRef attribute, any change to the value of the parameter in the called procedure does affect the value of the corresponding argument in the calling procedure.
8. The difference between a subroutine and a function in the use of the keyword Return is that in a subroutine the keyword Return appears by itself because no value is being returned, whereas in a function Return is followed by the value to be returned.
9. The two syntax options for a function returning a value are the Return statement and assigning to the function name the value to be returned.

10. Writing your own procedures enables you to organize your code in smaller, easier-to-read code blocks. Additionally, if you are performing essentially the same task from several places in the program, you can avoid duplication of code by putting the code that performs that task in one procedure, as opposed to repeating that code in each place in the program that may call for the performance of that task. Further, if you later have to fix a bug in how you perform that task, or simply find a better way to perform the task, you only have to change the code in one place rather than many.

Chapter 10

1. A message box is modal.
2. The Show method of the MessageBox class returns a member of the DialogResult enumeration corresponding to the button the user clicked.
3. No, you may call the Show method of the MessageBox class with a different number of arguments because that method is overloaded.
4. Yes, buttons in a message box automatically have a DialogResult value.
5. You would use the DialogResult data type for a variable you would use to store the return value of the Show method of the MessageBox class.
6. An enumeration is a list of related choices.
7. You use the ShowDialog method of the Form object to display a modal form.
8. The return value of showing a dialog form is the DialogResult property of that form.
9. No, buttons in a dialog form you create do not automatically have a DialogResult value; you have to assign a value to the DialogResult property of each button.
10. You use the Show method of the Form object to display a modal form.

Chapter 11

1. A main menu is represented by the MenuStrip class.
2. Each item on a main menu is represented by the ToolStripMenuItem class.
3. An access key is the keyboard combination of the ALT key plus a letter in the menu item that is underlined.

4. No, the Click event is raised only for menu items that do not have subsidiary menu items, because when a menu item with subsidiary items is clicked, the behavior is to display the subsidiary menu items.
5. You gray out a menu item so it is not available when it should not be by setting its Enabled property to False.
6. The Items collection of the MenuStrip component contains a collection of the ToolStripMenuItems belonging to the MenuStrip.
7. The shortcut or context menu is represented by the ContextMenuStrip class.
8. Each item on a context menu is represented by the ToolStripMenuItem class.
9. The Items collection of the ContextMenuStrip component contains a collection of the ToolStripMenuItems belonging to the ContextMenuStrip.
10. The different alternatives for having a context menu item's functionality handled by the corresponding main menu item are AddHandler, expanding the Handles clause, and calling another event procedure.

Chapter 12

1. The toolbar is represented by the ToolStrip class.
2. Each item on the toolbar is represented by the ToolStripItem class.
3. The Items collection of the ToolStrip component contains a collection of the ToolStripItems belonging to the ToolStrip.
4. No, a toolbar item is not limited to a button, but instead may be one of several other types of controls.
5. Toolbar buttons are immediately accessible, whereas menu items may be nested several levels deep and can be accessed only by multiple mouse clicks or keystrokes. Additionally, a toolbar button is visual, which gives a more visual interface than the text of a menu item.
6. Different alternatives for having a toolbar item's functionality handled by the corresponding main or context menu item are AddHandler, expanding the Handles clause, and calling another event procedure.
7. The DisplayStyle property of the ToolStripItem class determines whether an image or text may be displayed on a button.
8. The Image property of the ToolStripItem class determines the image displayed on a button.

9. The Items Collection Editor is useful in adding controls to a toolbar.
10. One good prefix for naming a toolbar button is “tbtn,” where “t” stands for *toolbar* and “btn” stands for *button*.

Chapter 13

1. The Open dialog box is a control of the OpenFileDialog class.
2. You use the ShowDialog method of the OpenFileDialog class to show an Open dialog box.
3. The return value of showing an Open dialog box is either DialogResult.OK if the user chose the Open button or DialogResult.Cancel if the user chose the Cancel button.
4. The OpenFileDialog class has a FileName property whose value is a string containing the path to and the name of the file selected in the Open dialog box.
5. The Save dialog box is a control of the SaveFileDialog class.
6. You use the ShowDialog method of the SaveFileDialog class to show a Save dialog box.
7. The return value of showing a Save dialog box is either DialogResult.OK if the user chose the Save button or DialogResult.Cancel if the user chose the Cancel button.
8. The SaveFileDialog class has a FileName property whose value is a string containing the path to and the name of the file to be saved.
9. You may use the StreamReader class to read from a text file.
10. You may use the StreamWriter class to write to a text file.

Chapter 14

1. A data provider is a code component used by your application to connect to a specific database format.
2. Server Explorer enables you to view and make changes to databases on your computer or on any other computer to which you have network access and permissions.

3. A table is a collection of data, usually on a particular subject such as customers, employees, and so on.
4. Each column in a table also may be called a field.
5. Each row in a table also may be called a record.
6. The code components used for database access in the .NET class library are referred to by the name ADO.NET.
7. The OleDbConnection class represents a connection to a data source.
8. You use the OleDbCommand class to execute commands to a database.
9. You use the OleDbDataAdapter class to package a database connection with a set of data commands.
10. A DataSet is a representation of the data stored in RAM.

Chapter 15

1. The code components used for web applications in the .NET class library are referred to by the name ASP.NET.
2. A URL, an acronym for Uniform Resource Locator, represents an address of a web page.
3. HTTP is an acronym for Hypertext Transfer Protocol. HTTP defines how messages are formatted and transmitted, and what actions web servers and browsers should take in response to various commands.
4. The .aspx extension indicates that the web page is part of an ASP.NET application.
5. A URL such as `http://localhost/localstart.asp` would be the virtual path to the web page, whereas a file path such as `C:\inetpub\wwwroot\localstart.asp` would be the physical path to the web page.
6. You may use the ASP.NET Web Site project template to create a web application.
7. GridView is the web control that corresponds to the DataGridView control used in Windows applications.
8. MapPath is the method of the HttpServerUtility class that returns the physical file path that corresponds to (is mapped to) the specified virtual path on a web server.

9. Page is the name of the class that is the web application equivalent of the Form class in a Windows application.
10. DataBind is the method of the GridView that needs to be called in a web application so the GridView will not be blank.

Final Exam

1. IDE is an acronym for Integrated Development Environment. The term “development environment” refers to Visual Basic 2005’s role as an application to assist you in developing applications. The term “integrated” means the tools to design your application and to write, test, and run your code are all together in one application.
2. A computer cannot do anything without step-by-step instructions from us telling it what to do. These instructions, written by a computer programmer, are called a computer program.
3. A programming language is used by computer programmers to write instructions for computers.
4. Machine language is a programming language that is understood by computers.
5. The term “higher level” means that a programming language such as Visual Basic 2005 is far closer to the structure and syntax of human language than to the ones and zeroes understood by a computer.
6. The term “lower level” means that a programming language such as machine language is far closer to the ones and zeroes understood by a computer than it is to the structure and syntax of human language.
7. In general, a compiler translates the code you write into corresponding machine language instructions. The compiler in Visual Basic 2005 translates the code into an intermediate language that then is translated into machine language.
8. Programming languages, including Visual Basic, use classes to represent a person, place, thing, or concept.
9. An object of a class is a single instance of a class, just like each of us could be said to be an object or instance of a Person class.
10. Namespaces are used to organize code in a logical manner.
11. A property is a characteristic or attribute of a class.

12. A Windows application has a graphical user interface (GUI) and is event-driven.
13. An event is something that happens to an object of a class, such as a result of user interaction.
14. An event procedure contains code that executes when a specific event happens to a specific object.
15. The purpose of the assignment operator is to assign the expression to its right to the variable or property to its left.
16. The purpose of the Toolbox is to display controls that you can add to your form.
17. You may add a control from the Toolbox on to your form either by double-clicking the control in the Toolbox or by dragging the control from the Toolbox and then dropping it on to the form.
18. The Name property of a control is used to identify that control in code.
19. A naming convention is a consistent method of naming, such as naming controls.
20. The text displayed by a Label may identify another, adjacent control, or it may display data.
21. A parameter represents information that is available to an event procedure.
22. A data type signifies whether the data is numeric, text, yes/no, and so forth.
23. The purpose of a variable is to store data of your choosing.
24. Yes, Visual Basic 2005 by the default setting of On for Option Explicit requires you to declare a variable before you refer to it in code.
25. A local variable is a variable declared inside of a procedure.
26. No, you do not have to assign a value to a variable when you declare it.
27. A constant's value cannot change during the life of the program, whereas a variable's value may change during the life of the program.
28. Yes, you have to assign a value to a constant when you declare it.
29. Operator precedence determines, when there are two or more arithmetic operators, which arithmetic operation is done first.
30. The Mod operator provides only the remainder resulting from division.
31. All arithmetic operators have precedence over the assignment operator.

32. The Parse method of the Integer class converts the string representation of an integer into actual integer values.
33. The ToString method of the Integer class converts an integer into its string representation.
34. A method is something an object of a class does.
35. The WriteLine method of the Debug class outputs a line to the Output window.
36. The data type of the result of a comparison performed by a comparison operator is Boolean (that is, True or False).
37. Arithmetic operators have higher precedence than comparison operators.
38. A logical operator is used to combine multiple comparisons.
39. Not is the logical operator that operates on only one operand rather than two.
40. Comparison operators have higher precedence than logical operators.
41. Modal means a form must be closed before the application user can return to any other form in the application.
42. The return value of the InputBox function if the OK button is clicked is whatever the user typed in the input box.
43. The return value of the InputBox function if the Cancel button is clicked is an empty string.
44. An exception is a problem that occurs while the program is executing that must be dealt with before the program can proceed.
45. The TryParse method of the Integer class converts the string representation of an integer into an integer, but also returns a Boolean value (True or False) indicating whether the conversion was successful.
46. The CheckBox and RadioButton controls.
47. The primary difference in the If...ElseIf statement and the Select Case control structure is that the If and ElseIf clauses each may evaluate completely different expressions, whereas a Select Case control structure may evaluate only one expression, which then must be used for every comparison.
48. A loop is a structure that repeats the execution of code until a condition becomes False.
49. A difference between the Do statement and the For...Next and While...End While statements is that a Do statement may test a condition at the bottom

as well as at the top of the statement, whereas the For...Next and While...End While statements may test a condition only at the top of the statement.

50. The For Each...Next loop executes the statement block for each element in a collection, instead of a specified number of times.
51. An array permits you to use a single variable to store multiple values.
52. The difference between declaring an array variable and a scalar variable is that with an array variable, unlike with a scalar variable, the array name is followed by a pair of parentheses, and within the parentheses you indicate the highest index of the array.
53. The lowest index of an array is zero.
54. The number of elements in an array is one greater than the highest index in that array because the index of the first element is zero.
55. A procedure is a block of one or more code statements that execute when called upon to do so.
56. A subroutine does not return a value, whereas a function does.
57. The Private access specifier limits access to the class in which the procedure was declared.
58. Calling a subroutine causes it to execute.
59. When a parameter has the ByVal attribute, any change to the value of the parameter in the called procedure does not affect the value of the corresponding argument in the calling procedure. By contrast, when a parameter has the ByRef attribute, any change to the value of the parameter in the called procedure does affect the value of the corresponding argument in the calling procedure.
60. Writing your own procedures enables you to organize your code in smaller, easier-to-read code blocks. Additionally, if you are performing essentially the same task from several places in the program, you can avoid duplication of code by putting the code that performs that task in one procedure, as opposed to repeating that code in each place in the program that may call for the performance of that task. Further, if you later have to fix a bug in how you perform that task, or simply find a better way to perform the task, you only have to change the code in one place rather than many.
61. A message box is modal.
62. The Show method of the MessageBox class returns a member of the DialogResult enumeration corresponding to the button the user clicked.
63. Yes, buttons in a message box automatically have a DialogResult value.

64. You would use the DialogResult data type for a variable you may use to store the return value of the Show method of the MessageBox class.
65. You use the ShowDialog method of the Form object to display a modal form.
66. The return value of showing a dialog form is the DialogResult property of that form.
67. No, buttons in a dialog form you create do not automatically have a DialogResult value; you have to assign a value to the DialogResult property of each button.
68. You use the Show method of the Form object to display a modeless form.
69. A main menu is represented by the MenuStrip class.
70. No, the Click event is raised only for menu items that do not have subsidiary menu items, because when a menu item with subsidiary items is clicked, the behavior is to display the subsidiary menu items.
71. You gray out a menu item so it is not available when it should not be by setting its Enabled property to False.
72. The Items collection of the MenuStrip component contains a collection of the ToolStripMenuItem belonging to the MenuStrip.
73. The shortcut or context menu is represented by the ContextMenuStrip class.
74. The Items collection of the ContextMenuStrip component contains a collection of the ToolStripMenuItem belonging to the ContextMenuStrip.
75. Different alternatives for having a context menu item's functionality handled by the corresponding main menu item include using AddHandler, expanding the Handles clause, and calling another event procedure.
76. The toolbar is represented by the ToolStrip class.
77. Each item on the main menu is represented by the ToolStripItem class.
78. The Items collection of the ToolStrip component contains a collection of the ToolStripItem belonging to the ToolStrip.
79. Toolbar buttons are immediately accessible, whereas menu items may be nested several levels deep and can be accessed only by multiple mouse clicks or keystrokes. Additionally, a toolbar button gives a more visual interface than the text of a menu item.
80. Different alternatives for having a toolbar item's functionality handled by the corresponding main or context menu item include using AddHandler, expanding the Handles clause, and calling another event procedure.

81. You use the ShowDialog method of the OpenFileDialog class to show an Open dialog box.
82. The return value of showing an Open dialog box is either DialogResult.OK if the user chose the Open button or DialogResult.Cancel if the user chose the Cancel button.
83. The OpenFileDialog class has a FileName property whose value is a string containing the path to and the name of the file selected in the Open dialog box.
84. You use the ShowDialog method of the SaveFileDialog class to show a Save dialog box.
85. The return value of showing a Save dialog box is either DialogResult.OK if the user chose the Save button or DialogResult.Cancel if the user chose the Cancel button.
86. The SaveFileDialog class has a FileName property whose value is a string containing the path to and the name of the file to be saved.
87. You may use the StreamReader class to read from a text file.
88. You may use the StreamWriter class to write to a text file.
89. A data provider is a code component used by your application to connect to a specific database format.
90. A table is a collection of data, usually on a particular subject, such as customers, employees, and so on.
91. Each column in a table also may be called a field.
92. Each row in a table also may be called a record.
93. The code components used for database access in the .NET class library are referred to by the name ADO.NET.
94. A DataSet is a representation of the data stored in RAM.
95. The code components used for web applications in the .NET class library are referred to by the name ASP.NET.
96. A URL, an acronym for Uniform Resource Locator, represents an address of a web page.
97. HTTP is an acronym for Hypertext Transfer Protocol. HTTP defines how messages are formatted and transmitted, and what actions web servers and browsers should take in response to various commands.



98. A URL such as `http://localhost/localstart.asp` would be the virtual path to the web page, whereas a file path such as `C:\Inetpub\Wwwroot\localstart.asp` would be the physical path to the web page.
99. `MapPath` is the method of the `HttpServerUtility` class that returns the physical file path that corresponds to (is mapped to) the specified virtual path on a web server.
100. `Page` is the name of the class that is the web application equivalent of the `Form` class in a Windows application.



INDEX

Numbers

- 0-9, ASCII values of, 103
- :1040 after localhost, significance of, 289

Symbols

- (subtraction operator), using, 81
- # (pound sign) pattern-matching character, using in string comparisons, 104-105
- & (ampersand)
 - preceding access keys with, 205
 - using, 81
- () (parentheses), using with event procedures, 56-57
- (_) line-continuation character, using
 - with event procedures, 56
- * (asterisk) pattern-matching character, using in string comparisons, 104-105
- * (multiplication operator), using, 81
- *= arithmetic/assignment operator, uses of, 84
- / (division operator), using, 82
- /= arithmetic/assignment operator, uses of, 84
- ? (question mark) pattern-matching character, using in string comparisons, 104-105
- \ (division operator), using, 82
- \= arithmetic/assignment operator, uses of, 84
- ^ (exponent operator), using, 81-82
- ^= arithmetic/assignment operator, uses of, 84
- + (addition operator), using, 80-81
- += arithmetic/assignment operator, uses of, 84
- < (less than operator), result of, 100
- <= (less than or equal to operator), result of, 100
- <> (inequality operator), result of, 101

- = (assignment operator), using with event procedures, 39-40
- = (equality operator), result of, 101
- = arithmetic/assignment operator, uses of, 84
- > (greater than operator), result of, 101
- >= (greater than or equal to operator)
 - result of, 101
 - using Or logical operator with, 108
- ' (apostrophe), using for comments, 40
- " (quotation marks), using with strings and event procedures, 40

A

- Abort choice, using with DialogResult enumeration, 186
- AbortRetryIgnore choice, using with MessageBoxButtons enumeration, 184
- About dialog box as dialog form, significance of, 180
- access keys, using with menu items, 205-206
- Accessibility element
 - of functions, 169
 - of subroutines, 160, 161
- Add Connection dialog box, using with Northwind Traders database, 258-259
- Add New Item dialog box, using with dialog forms, 189
- Add Reference dialog box, using with database project, 266-267
- AddHandler keyword, using with context menu items, 216, 219
- addition operator (+), using, 80-81
- ADO.NET classes, using with database project, 266

- algorithm, creating for Change Machine project, 90-92
- ALT key, using with menu items, 205-206
- ampersand (&)
 - preceding access keys with, 205
 - using, 81
- And logical operator
 - versus AndAlso operator, 111
 - overview of, 106-107
- AndAlso logical operator
 - versus And operator, 111
 - overview of, 107
- apostrophe ('), using for comments, 40
- applications, viewing, 23. *See also* Windows applications
- arguments
 - matching with parameters, 165-166
 - passing to procedures, 164-168
- arithmetic operators
 - addition operator (+), 80-81
 - combining with assignment operators, 83-84
 - division operators (/ and \), 82
 - exponent operator (^), 81-82
 - multiplication operator (*), 81
 - precedence of, 83
 - subtraction operator (-), 81
- arrays. *See also* variables
 - declaring, 153
 - default values for, 153-154
 - description of, 153
 - dimensions of, 155
- As Type element
 - description of, 169
 - using with functions, 169
 - using with ReturnInput function, 170
- ASCII values of commonly used characters, 103

- ASP.NET, overview of, 278
 - ASP.NET application IDE, significance of, 290–291
 - ASP.NET Development Server, using, 288–290
 - ASP.NET support, including in websites, 287
 - assignment operator (=)
 - combining arithmetic operators with, 83–84
 - using with event procedures, 39–40
 - asterisk (*) pattern-matching character, using in string comparisons, 104–105
 - Asterisk choice, using with MessageBoxIcon enumeration, 185
 - auto hide behavior, impact on Toolbox, 46
 - AutoSize property, using with Label controls, 48–49
 - A-Z and a-z, ASCII values of, 103
-
- B**
- BackColor property, changing for labels, 53
 - bitmap files, using as toolbar images, 230–231
 - blnResident prefix, using with data types, 72
 - Boolean data types, using, 67
 - Boolean values, returning for functions, 173–174
 - btnClose, adding code to Click event of, 181
 - btnNewCaption, adding code to Click event of, 192, 194
 - btnWrite button, adding to form, 245
 - Build menu
 - options on, 13
 - running projects from, 12
 - Button1-3 MessageBoxDefaultButton enumerations, descriptions of, 185
 - buttons. *See* toolbar buttons
 - ByVal and ByRef attributes, using with procedures, 166–167
-
- C**
- “c” argument, using as format specifier in Pizza Calculator, 134
 - Cancel button
 - determining selection of, 242–243
 - effect on input boxes, 119
 - Cancel choice, using with DialogResult enumeration, 186
 - Change Machine project
 - creating, 87–89
 - creating algorithm for, 90–92
 - description of, 86
 - Char data types, using, 68
 - characters, ASCII values of, 103
 - CheckBox controls
 - versus RadioButton controls, 130
 - using with If control structures, 128–130
 - Checked property, using with CheckBox control, 129
 - Choose Data Source dialog box, using with Northwind Traders database, 257–258
 - class methods, significance of, 86
 - classes
 - and events, 35
 - and properties, 28–29
 - use of, 25–26
 - Clear method of DataSet object, effect of, 273
 - Click event procedure
 - of btnCalculate, 89
 - of btnClear, 89
 - of btnClose, 181
 - of btnNewCaption, 192, 194
 - of Calculate button in Pizza Calculator, 133
 - calling for context menu items, 216
 - calling PrintInput subroutine from, 163
 - handling for Cut toolbar button, 233
 - of Paste toolbar button, 234
 - of Read button, 244, 248–249
 - using with menu items, 207–208
 - of Write button, 250–252
 - Close method of form object, significance of, 187
 - code
 - associating with clicks of toolbar buttons, 233–234
 - identifying errors in, 57–59
 - translating with compilers, 16
 - writing inside event procedures, 38–39
 - code view
 - implementing, 23
 - for web forms, 290–291
 - commands, creating for database project, 269–271
 - CommandText property, using with database project, 270
 - comments, indicating with apostrophe (’), 40
 - comparison operators
 - capabilities of, 98
 - and For..Next statements, 141
 - numeric comparison operators, 100–101
 - overview of, 100
 - precedence of, 105
 - string comparison operators, 102
 - comparisons
 - combining with logical operators, 106
 - using Like operator with, 104–105
 - using Option Compare statement with, 103
 - using pattern matching with, 104
 - compiler errors. *See also* errors
 - “Name ‘intVar’ is not declared,” 73
 - “Option Strict On disallows implicit conversions...,” 94
 - compilers, purpose of, 16
 - computer program, definition of, 15
 - connections
 - linking commands to, 270–271
 - opening for database project, 269
 - ConnectionString property of OleDbConnection class
 - using with database project, 268
 - using with database web applications, 296
 - console, example of, 32
 - constants
 - assigning values to, 76
 - declaring, 75–76
 - declaring for Pizza Calculator project, 132–133
 - guidelines for use of, 76–77
 - context menu items
 - adding functionality to, 214–216
 - using AddHandler keyword with, 216, 219
 - using handles clause with, 216
 - context menus, features of, 209–210. *See also* menus

- ContextMenuStrip objects
 - adding menu items to, 211–214
 - adding to forms, 210–211
 - pasting items into, 214–215
 - control structures
 - If control structures, 119–123
 - Select Case, 134–137
 - controls
 - copying from Toolbox to forms, 46
 - relocating, 46–48
 - resizing, 48–49
 - coordinates in graphing, relationship to Label controls, 52–53
 - Copy command, duplicating in context and main menus, 213–214
 - Copy toolbar button, assigning image to, 232
 - copying menus, 214
 - counter variables, using Step statements with, 144
 - CTRL shortcut key, using with menu items, 206–207
 - Customers table in Northwind Traders database, features of, 262–263
 - Cut command, duplicating in context and main menus, 213–214
 - Cut toolbar button
 - assigning image to, 232
 - using Click event procedure with, 233
-
- D**
- data namespaces, importing for database project, 266
 - data sources, connecting to, 268
 - data types
 - numeric data types, 67
 - overview of, 66
 - prefixes for, 71
 - text data types, 68
 - of Visual Basic properties, 68–69
 - database project. *See also* Northwind Traders database; projects
 - creating command for, 269–271
 - creating connection for, 267–269
 - creating form for, 264–265
 - description of, 264
 - filling DataGridView control in, 271–274
 - importing data namespaces for, 266
 - database web applications. *See also* web applications
 - adding code, 297–298
 - adding GridView control to, 292–295
 - databases
 - locating on web servers, 295–297
 - tables in, 261
 - DataBind method of GridView, relationship to database web applications, 298
 - DataGridView control
 - connecting to DataSet, 274
 - filling, 271–274
 - using with database project, 265
 - DataSet class
 - connecting DataGridView to, 274
 - creating for database project, 272–273
 - dblGPA prefix, using with data types, 72
 - Debug menu, running projects from, 39
 - debugging
 - overview of, 98–99
 - projects, 125
 - Debug.WriteLine statement. *See* WriteLine method of Debug class
 - Default Web Site, starting for IIS, 282–283, 286
 - design time
 - changing properties at, 31
 - explanation of, 13, 22
 - designer view
 - choosing and implementing, 23
 - displaying Object Browser from, 25
 - opening Properties window from, 29–30
 - dialog forms. *See also* forms; modal versus modeless forms
 - About dialog box as, 180
 - accessing values from, 194
 - changing properties of, 190–191
 - creating project for, 188–192
 - versus MessageBox class, 192, 193
 - preventing from closing, 193–194
 - showing and returning results of, 192–193
 - DialogResult values
 - processing, 187–188
 - returning, 193–194
 - returning with ShowDialog method, 242
 - Dim access specifier, using with variables, 70
 - disconnected application versus persistent connection, 267–268
 - DisplayStyle property, setting for images on toolbar buttons, 228
 - division operators (/ and \), using, 82
 - Do statements
 - testing conditions at bottom of, 151–152
 - testing conditions at top of, 150–151
 - DOS versus Windows applications, 32–34
 - Double class, Parse method of, 85
 - Double data types, using, 67
 - DropDownItems collection property, description of, 203
-
- E**
- Else clauses, characteristics of, 123
 - Enabled property of menu items, setting to False, 209
 - End Function element, description of, 169
 - End If, omission from If...Then statements, 120
 - End Sub element of subroutines, description of, 160
 - environment setting defaults, choosing, 7–8
 - equality operator (=), result of, 101
 - Error choice, using with MessageBoxIcon enumeration, 185
 - errors, identifying in code, 57–59. *See also* compiler errors
 - event procedure stubs
 - creating, 36–38
 - creating for Click event of btnCalculate, 89
 - creating for Click event of btnClear, 89
 - creating for MouseMove form event, 54
 - event procedures
 - creating for database web applications, 297
 - detection of, 36
 - Handles clause in, 57
 - line-continuation character in, 56
 - parameters of, 56–57

purpose of, 35
writing code inside of, 38–39

events, relationship to Windows programming, 35

exceptions
determining occurrence of, 126–127
overview of, 124–125
unhandled exceptions, 125–126

Exclamation choice, using with
 MessageBoxIcon enumeration, 185

Exit For statements, using, 145

Exit Function element, description of, 169

Exit Sub statements, using with subroutines, 160, 162

Exit While statements, using, 149–150

expander, location of, 25

explicit versus implicit type
conversion, 93

exponent operator (^), using, 81–82

expressions, using with Select Case control structure, 135

F

fields, displaying in Customers table, 262–263

File menu item, adding, 201

FileName property of OpenFileDialog control
using with database web applications, 296
value of, 244

files, identifying for opening, 243–244

Fill method of OleDbDataAdapter object, effect of, 273

Find form, modeless type of, 117

firefighter analogy, applying to calling subroutines, 162–163

For Each...Next loops, using, 152

For...Next statements
and comparison operators, 141
execution of, 143
nesting If...End If statements in, 146
and Step statements, 145
syntax of, 140–141
versus While...End While loops, 148–150

Form class, relationship to System.Windows.Forms namespace, 27–28

Form object
Close method of, 187
ShowDialog method of, 192–193

Form1 class events
choosing, 36
listing, 37

format specifier, using in Pizza Calculator project, 134

forms. *See also* dialog forms; helper forms; modal versus modeless forms; web forms
adding ontextMenuStrip objects to, 210–211
adding labels to, 53
adding MenuStrip controls to, 199–200
adding OpenFileDialog control to, 240–241
adding SaveFileDialog class to, 244–245
adding toolbars to, 222–223
copying controls to, 46
creating for database project, 264–265
displaying, 192
displaying in designer view, 23–24
modal versus modeless types of, 194–195

Friend accessibility specifier, meaning of, 161

frmCaption object, creating, 193

Function element, description of, 169

functions
calling, 170–171
in databases, 262
declaring, 168–170
definition of, 158
elements of, 169
returning Boolean values for, 173–174
returning default values for, 173
returning values for, 171–174
versus subroutines, 168–169

G

GetLowerBound and GetUpperBound methods, using with arrays, 154

greater than operator (>), result of, 101

greater than or equal to operator (>=)
result of, 101
using Or logical operator with, 108

GridView controls
adding to database web applications, 292–295
DataBind method of, 298

GUIs (graphical user interfaces), Windows applications as, 32–34

H

Hand choice, using with
 MessageBoxIcon enumeration, 185

Handles clause
expanding for Click event procedure, 233–234
using with context menu items, 216
using with event procedures, 57

hard drive space, requirements for, 5

helper forms, example of, 179–180.
See also forms

HttpServerUtility class, using with database web applications, 296

I

IDE (Integrated Development Environment)
for ASP.NET application, 290
displaying, 11
running projects with, 12–13

If control structures
RadioButton controls, 130
using CheckBox controls with, 128–130
using in Pizza Calculator project, 133–134
varieties of, 119–120

If...Else statements, using with
 CheckBox controls, 130

If...ElseIf statements
versus Select Case control structure, 137
using, 122–123
using TryParse method with, 128
using with RadioButton controls, 130

If...End If statements, nesting in For...Next statements, 146

- If...Then statements, using, 120–121
 - If...Then...Else statements
 - using, 121–122
 - using with Exit For statements, 145
 - using with functions and Boolean values, 173
 - Ignore choice, using with DialogResult enumeration, 186
 - IIS (Internet Information Services)
 - determining installation of, 279
 - installing, 280
 - opening Default Web Site for, 282–283
 - overview of, 278
 - IIS Admin Service, starting, 280–282
 - images, associating with toolbar buttons, 227–232
 - ImageScaling property, setting to SizeToFit, 233
 - implicit versus explicit type conversion, 93
 - Imports statements, using with database web applications, 297
 - inequality operator (<>), result of, 101
 - Information choice, using with MessageBoxIcon enumeration, 185
 - inheritance by classes, explanation of, 27
 - initialization, relationship to local variables, 73
 - “Input string was not in a correct format” message, displaying, 124–125
 - input validation
 - and exceptions, 124–128
 - overview of, 124
 - InputBox function, using, 116–119
 - installing Visual Basic 2005, 6–7
 - Integer class, TryParse method of, 127–128
 - Integer data types, using, 67
 - intScore prefix, using with data types, 72
 - intVar variable, declaring, 73, 74
 - “Invalid score,” outputting with Select Case control structure, 136–137
 - Is keyword, using with Select Case control structure, 135
 - Items Collection Editor
 - adding items to context menus with, 212–213
 - opening, 203, 225
 - using with images for toolbar buttons, 228–229
 - using with toolbar buttons, 224
 - Items collection, using with MenuStrip component, 202–204
-
- K**
- keyboard shortcuts, using with menu items, 206–207
-
- L**
- Label class, properties of, 50–51
 - Label controls
 - situating in forms, 46
 - using, 52–55
 - labels, adding for forms, 53
 - less than operator (<), result of, 100
 - less than or equal to operator (<=), result of, 100
 - Like operator, using with comparisons, 104–105
 - line-continuation character (_), using with event procedures, 56
 - local scope, declaring constants at, 75
 - local variables, declaring, 72–73
 - logical operators
 - AndAlso logical operator, 107
 - AndAlso versus And operator, 111
 - combining comparisons with, 106
 - For Each...Next loops, 152
 - Not operator, 110
 - And operator, 106–107
 - Or operator, 108
 - OrElse operator, 109
 - OrElse versus Or operator, 111
 - overview of, 106
 - precedence of, 110–111
 - using If...Then statements with, 121
 - Xor operator, 109–110
 - loops
 - and compared values, 142
 - definition of, 142
 - Do statements, 150–152
 - execution of, 142–143
 - Exit For statements, 145
 - For...Next statements, 140–145
 - and iteration, 144
 - nesting, 146–147
 - and Step statements, 144–145
 - using with arrays, 153–154
 - While...End While statements, 147–148
-
- M**
- machine language, explanation of, 15
 - main menus. *See also* menus
 - versus context menus, 212
 - creating, 198–199
 - MapPath method of HttpServerUtility class, using with database web applications, 296
 - .mdb extension, explanation of, 256
 - Me keyword, using with dialog forms, 194
 - menu items
 - adding functionality to, 207–208
 - adding to MenuStrip components, 200–201
 - adding to ContextMenuStrip objects, 211–214
 - deleting, 202
 - disabling, 208–209
 - naming, 212
 - recalling, 202
 - setting Text properties for, 201
 - using access keys with, 205–206
 - using separator bars with, 207
 - using shortcut keys with, 206–207
 - menus. *See also* context menus; main menus
 - copying, 214
 - hiding, 209
 - MenuStrip class
 - copying items from, 214
 - significance of, 198
 - using Items collection with, 202–204
 - MenuStrip controls, adding to forms, 199–200
 - message boxes
 - creating project for, 181–182
 - features of, 180–181
 - as helper forms, 179–180
 - modal aspect of, 182
 - using Show method with, 182–186

- MessageBox class versus dialog forms, 192, 193
 - MessageBoxButtons parameter of Show method
 - description of, 183
 - syntax of, 184
 - MessageBoxDefaultButton parameter of Show method
 - description of, 183
 - overview of, 185–186
 - MessageBoxIcon parameter of Show method
 - description of, 183
 - overview of, 184–185
 - methods
 - definition of, 86
 - as procedures, 159
 - Mod operator, using, 82
 - modal versus modeless forms, 116–117, 182. *See also* forms; dialog forms
 - module-level scope, declaring constants at, 75
 - module-level variables, declaring, 73–74
 - mouse coordinates, relationship to Label controls, 52–53
 - MouseEventArgs class, properties of, 57
 - MouseMove form event, creating event procedure stub for, 54
 - multiplication operator (*), using, 81
-
- N**
- Name element
 - of functions, 169
 - of subroutines, 160
 - Name property of Label class, features of, 51
 - namespaces
 - explanation of, 27–28
 - importing for database project, 266
 - nesting, examples of, 146–147
 - .NET Framework, significance of, 28
 - New keyword, using with StreamReader class, 248
 - New Project dialog box, displaying, 9
 - New Web Site dialog box, displaying, 287
 - No choice, using with DialogResult enumeration, 186
 - None choice
 - using with DialogResult enumeration, 186
 - using with MessageBoxIcon enumeration, 185
 - Northwind Traders database. *See also* database project
 - connecting to, 257–259
 - Customers table in, 262–263
 - obtaining and installing, 256
 - Not logical operator, using, 110
 - numbers, converting to string representations, 85–86
 - numeric comparison operators, using, 100–101
 - numeric data types, overview of, 67
 - nwind.mdb file, finding and choosing, 259
-
- O**
- Object Browser, displaying from designer view, 25
 - objects
 - and properties, 28–29
 - use of, 25–26
 - OK button, effect on input boxes, 118
 - OK choice
 - using with DialogResult enumeration, 186
 - using with MessageBoxButtons enumeration, 184
 - OKCancel choice, using with MessageBoxButtons enumeration, 184
 - OleDbCommand object, instantiating, 269
 - OleDbConnection class, using with database project, 266, 268–269
 - OleDbDataAdapter, creating for database project, 271–272
 - On, setting Option Strict to, 93–94
 - Open button, determining selection of, 242–243
 - Open dialog box, displaying in Notepad and DOS text editor, 33
 - Open method of OleDbConnection object, using with database project, 269
 - Open Project dialog box, opening, 21
 - OpenFileDialog class, FileName property of, 244, 296
 - OpenFileDialog control
 - adding to forms, 240–241
 - showing, 241–242
 - using with database project, 265
 - operating system, requirements for, 5
 - operator precedence, explanation of, 83
 - operators. *See* arithmetic operators; comparison operators; logical operators
 - Option Compare statement, using with comparisons, 103
 - Option Strict, setting to On, 93–94
 - Or logical operator
 - versus OrElse operator, 111
 - overview of, 108
 - OrElse logical operator
 - versus Or operator, 111
 - using, 109
 - Output window, outputting WriteLine method to, 99
-
- P**
- Page_Load event procedure, coding for database web applications, 297
 - Parameter list element
 - of functions, 169
 - of subroutines, 160
 - parameters
 - of InputBox function, 118
 - matching with arguments, 165–166
 - naming for use with procedures, 164
 - of Power subroutine, 164
 - and procedures, 162, 163–164
 - of Show method, 182–183
 - using with event procedures, 56–57
 - parent menu item, example of, 198–199
 - parentheses (()), using with event procedures, 56–57
 - Parse method of Integer class
 - relationship to type conversions, 92
 - using, 85
 - Paste command, duplicating in context and main menus, 213–214
 - Paste toolbar button
 - assigning image to, 232
 - using Click event procedure with, 234
 - pattern matching, using with comparisons, 104
 - persistent connection versus disconnected application, 267–268

- physical and virtual paths, relationship to URLs, 285–286
 - Pizza Calculator Input box, displaying, 116
 - Pizza Calculator project
 - calculating price in, 133–134
 - code for, 132–133
 - creating, 131–132
 - functionality of, 132
 - restoring to initial settings, 134
 - port numbers, identifying for ASP.NET Development Server, 289
 - pound sign (#) pattern-matching character, using in string comparisons, 104–105
 - Power function, calling, 171–172
 - Power subroutine
 - calling, 164–165
 - converting to Power function, 170–171
 - parameter of, 164
 - precedence
 - of arithmetic operators, 83
 - of comparison operators, 105
 - of logical operators, 110–111
 - prefixes for data types, examples of, 71–72
 - Print dialog box
 - interacting with, 180
 - relationship to Label controls, 50
 - PrintInput subroutine
 - calling from Click event procedure, 163
 - elements of, 160–161
 - Private accessibility specifier, meaning of, 161
 - procedural programming, explanation of, 34–35
 - procedures. *See also* subroutines
 - accessibility specifiers for, 161
 - built-in versus programmer-defined procedures, 158–159
 - procedures, declaring variables inside of, 72
 - declaring with single and multiple parameters, 165
 - definition of, 157
 - methods as, 159
 - naming, 161–162
 - and parameters, 163–164
 - passing arguments to, 164–168
 - writing, 174–175
 - processors, requirements for, 5
 - programming languages, examples of, 15–16
 - project types, displaying, 9
 - projects. *See also* database project
 - Change Machine project, 86–92
 - debugging, 125
 - dialog forms, 188–192
 - message box, 181–182
 - naming and specifying locations of, 10–11
 - opening, 21
 - Pizza Calculator, 131–134
 - running, 12–13
 - running from Debug menu, 39
 - stopping, 125
 - Text Editor, 217–219
 - properties
 - changing at design time, 31
 - changing for dialog forms, 190–191
 - and data types, 68–69
 - displaying for buttons, 227
 - ImageScaling, 233
 - overview of, 28–29
 - relationship to class methods, 86
 - for ToolTipText, 228
 - Properties window
 - for ContextMenuStrip, 212
 - displaying Items collection of MenuStrip in, 202
 - displaying shortcut key options in, 207
 - opening, 29–30
 - showing DropDownItems collection of File menu item in, 204
 - Protected accessibility specifier, meaning of, 161
 - Protected Friend accessibility specifier, meaning of, 161
 - Public accessibility specifier, meaning of, 161
-
- Q**
- Question choice, using with MessageBoxIcon enumeration, 185
 - question mark (?) pattern-matching character, using in string comparisons, 104–105
 - quotation marks (=), using with strings and event procedures, 40
-
- quotient, displaying for Change Machine project, 91
-
- R**
- RadioButton controls, using with If control structures, 130
 - RAM (random access memory), requirements for, 5
 - Read button
 - Click event procedure for, 248–249
 - completing Click event procedure for, 249
 - modifying Click event procedure of, 244
 - ReadToEnd method of StreamReader class, description of, 248
 - Rebuild Solution option on Build menu, explanation of, 13
 - records in databases, definition of, 263
 - resizing controls, 48–49
 - Retry choice, using with DialogResult enumeration, 186
 - RetryCancel choice, using with MessageBoxButtons enumeration, 184
 - Return element
 - of functions, 169
 - of subroutines, 160
 - Return statements
 - using with functions, 172–173
 - using with Power function, 172
 - using with subroutines, 162
 - Return values
 - effect on input boxes, 118–119
 - of Show method, 186–188
 - ReturnInput function, using As Type statement with, 170
 - run time, explanation of, 13, 22
 - running totals, obtaining with arrays and loops, 154
-
- S**
- sales tax rates, declaring as constants, 76–77
 - SaveFileDialog class, adding to forms, 244–245
 - Select Case control structure versus If...ElseIf statements, 137
 - and Is keyword, 135
 - overview of, 134–135

- syntax of, 135
 - using, 136–137
 - Select Resource dialog box, assigning images to forms from, 229, 231
 - SELECT statements, using with database project, 269–270
 - separator bars, using with menu items, 207
 - Server Explorer, using with databases, 260–263
 - SHIFT shortcut key, using with menu items, 206–207
 - shortcut keys, using with menu items, 206–207
 - shortcut menus, features of, 209–210. *See also* menus
 - Show method
 - using Return value of, 186–188
 - using with message boxes, 182–186
 - ShowDialog method
 - of Form object, 192–193
 - of OpenFileDialog class, 242
 - using return value of, 243
 - SizeToFit setting, using with ImageScaling property, 233
 - .sln extension, explanation of, 22
 - smart task arrow, appearance of, 222, 225
 - Solution Explorer
 - displaying, 22
 - using with dialog forms, 190
 - solutions
 - building and rebuilding, 12–13
 - contents of, 22
 - SQL statements, using with database project, 269–270
 - Start page, displaying, 7–8
 - Statements element
 - of functions, 169
 - of subroutines, 160
 - Step statements
 - and loops, 144–145
 - role in For...Next statements, 141
 - Stop choice, using with MessageBoxIcon enumeration, 185
 - stored procedures in databases, definition of, 262
 - StreamReader class, reading from text files with, 246–248
 - StreamReader variables, instantiating, 247–248
 - StreamWriter class, writing to text files with, 250–251
 - string comparison operators, overview of, 102
 - String data types, using, 68
 - string representations of integers, converting to Integer values, 85
 - strName prefix, using with data types, 72
 - stubs, creating for event procedures, 36–38
 - Sub element of subroutines, description of, 160
 - subroutines. *See also* procedures
 - arguments of, 165
 - calling, 162–163, 164–165
 - declaring, 159–162
 - definition of, 158
 - versus functions, 168–169
 - using Return and Exit Sub statements with, 162
 - subtraction operator (–), using, 81
 - system requirements for Visual Basic 2005, 5–6
 - System.Data namespace
 - importing for database web applications, 297
 - using with database project, 266
 - System.IO namespace, importing for StreamReader class, 247
-
- T**
- tables in databases, definition of, 261
 - templates, displaying, 9
 - “The test score is valid,” displaying in Output window, 123
 - text data types, using, 68
 - Text Editor project
 - components of, 219
 - creating, 217–218
 - testing ShowDialog code in, 242
 - text files
 - closing, 249–250, 252–253
 - displaying contents of, 246
 - limitations of, 255
 - reading from, 246–248
 - reading into TextBox control, 248–249
 - writing to, 250–251
 - Text parameter of Show method, description of, 183
 - Text property
 - of CheckBox control, 129
 - of Label class, 50–51
 - setting for menu items, 201
 - using with dialog forms, 194
 - TextBox class
 - methods of, 219
 - writing to text file to, 251
 - “This line will always print,” printing to Output window, 145
 - Title parameter of Show method, description of, 183
 - toolbar buttons
 - adding, 224–227
 - associating code with clicks of, 233–234
 - associating images with, 227–232
 - setting ToolTipText property for, 228
 - toolbars, adding to forms, 222–223
 - Toolbox
 - adding GridView controls to, 293–294
 - copying controls from, 46
 - displaying, 44
 - expanding categories of, 45
 - unhiding, 46
 - ToolStrip objects, adding to forms, 222–223
 - ToolStripMenuItem objects
 - adding to ContextMenuStrip, 213
 - adding to MenuStrip component, 203
 - inclusion in MenuStrip objects, 198
 - ToolTipText property, setting for buttons, 228
 - ToString method
 - using, 85–86, 91, 94
 - using with procedures, 164
 - totals, obtaining running totals with arrays and loops, 154
 - True conditions, using If...Then statements with, 120–121
 - TryParse method of Integer class, using, 127–128
 - type conversions, overview of, 92–94

-
- U**
- unhandled exceptions, explanation of, 125–126
 - Until keyword, using with Do statements, 150–151
 - URLs (Uniform Resource Locators), relationship to web applications, 284–286
-
- V**
- values
 - accessing from dialog forms, 194
 - assigning to array elements, 153–154
 - assigning to constants, 76
 - comparing in loops, 142
 - options for returning of, 172–173
 - returning for functions, 171–174
 - variables. *See also* arrays
 - declaring, 70, 72–74
 - declaring inside procedures, 72
 - naming, 70–72
 - View menu, options on, 23
 - views in databases, definition of, 262
 - virtual and physical paths, relationship to URLs, 285–286
 - Visible property, setting for menus, 209
 - Visual Basic 2005
 - choosing version of, 6
 - installing, 6–7
 - starting, 7
 - system requirements for, 5–6
 - Visual Basic 2005 projects. *See* projects
-
- W**
- Warning choice, using with
 - MessageBoxIcon enumeration, 185
 - web applications. *See also* database web applications
 - ASP.NET, 278
 - creating, 287–291
 - IIS (Internet Information Services), 278–283
 - and URLs (Uniform Resource Locators), 284–286
 - web forms, design and code views of, 290–291. *See also* forms
 - web servers
 - computers as, 284–285
 - locating databases on, 295–297
 - website for Visual Basic 2005 editions, 6
 - While keyword, using with Do statements, 150
 - While...End While loops
 - versus For...Next statements, 148–150
 - overview of, 147–148
 - Windows applications. *See also* applications
 - versus DOS applications, 32–34
 - event-driven aspect of, 34–35
 - explanation of, 31–32
 - GUI (graphical user interface) of, 32–34
 - Windows Components Wizard, checking installation of IIS with, 279
 - Write button, Click event procedure of, 250, 251, 252
-
- Write method of StreamWriter class, effect of, 251
 - WriteLine method of Debug class
 - example of, 99, 101
 - relationship to loops, 142, 143
 - using with functions, 171
 - using with If...Then statements, 121
 - using with subroutines, 163
-
- X**
- X and Y coordinates, displaying for mouse pointer, 52–53, 56–57
 - Xor logical operator, using, 109–110
-
- Y**
- Yes choice, using with DialogResult enumeration, 186
 - YesNo choice, using with
 - MessageBoxButtons enumeration, 184
 - YesNoCancel choice, using with
 - MessageBoxButtons enumeration, 184
 - “You didn’t enter anything” output, receiving, 173–174
 - “You entered a positive number,” displaying in Output window, 120
 - “You entered a valid test score (0-100),” displaying in Output window, 122
 - “You entered something” output, receiving, 174

The fast and easy way to understanding computing fundamentals

- No formal training needed
- Self-paced, easy-to-follow, and user-friendly
- Amazing low price



0-07-225454-8



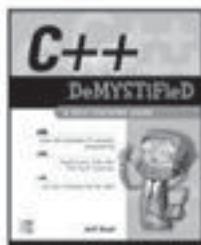
0-07-225363-0



0-07-225514-5



0-07-225359-2



0-07-225370-3



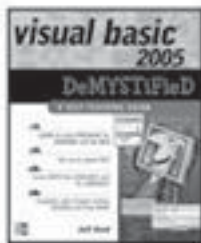
0-07-225364-9



0-07-225878-0



0-07-226134-X



0-07-226171-4



0-07-226170-6



0-07-226141-2



0-07-226182-X



0-07-226224-9



0-07-226210-9

For more information on these and other McGraw-Hill/Osborne titles, visit www.osborne.com.

OSBORNE DELIVERS RESULTS!

Mc
Graw
Hill

Osborne