

MICROSOFT®

# VISUAL C#® 2012

AN INTRODUCTION TO  
OBJECT-ORIENTED PROGRAMMING

JOYCE FARRELL



FIFTH EDITION

**Microsoft® Visual C#® 2012:  
An Introduction to Object-Oriented  
Programming, Fifth Edition**  
Joyce Farrell

**Executive Editor:** Kathleen McMahon  
**Senior Product Manager:** Alyssa Pratt  
**Development Editor:** Dan Seiter

**Editorial Assistant:** Sarah Ryan

**Brand Manager:** Kay Stefanski

**Content Project Manager:**  
Jen Feltri-George

**Art Director:** Cheryl Pearl, GEX

**Cover Designer:** Lisa Kuhn/Curio Press,  
LLC, [www.curiorepress.com](http://www.curiorepress.com)

**Cover Photo:** ©Leigh Prather/Veer  
Incorporated

**Print Buyer:** Julio Esperas

**Copyeditor:** Michael Beckett

**Proofreader:** Brandy Lilly

**Indexer:** Sharon Hilgenberg

**Compositor:** Integra Software  
Services Pvt. Ltd.

© 2014 Course Technology, Cengage Learning

ALL RIGHTS RESERVED. No part of this work covered by the copyright herein may be reproduced, transmitted, stored or used in any form or by any means—graphic, electronic, or mechanical, including but not limited to photocopying, recording, scanning, digitizing, taping, Web distribution, information networks, or information storage and retrieval systems, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act—without the prior written permission of the publisher.

For product information and technology assistance, contact us at  
**Cengage Learning Customer & Sales Support**, [www.cengage.com/support](http://www.cengage.com/support)

For permission to use material from this text or product,  
submit all requests online at [cengage.com/permissions](http://cengage.com/permissions)

Further permissions questions can be e-mailed to  
[permissionrequest@cengage.com](mailto:permissionrequest@cengage.com)

Library of Congress Control Number: 2013931000

ISBN-13: 978-1-285-09633-9

ISBN-10: 1-285-09633-9

**Course Technology**  
20 Channel Center Street  
Boston, MA 02210

Some of the product names and company names used in this book have been used for identification purposes only and may be trademarks or registered trademarks of their respective manufacturers and sellers.

Any fictional data related to people, companies, or URLs used throughout this book is intended for instructional purposes only. At the time this book was printed, any such data was fictional and not belonging to any real people or companies.

Course Technology, a part of Cengage Learning, reserves the right to revise this publication and make changes from time to time in its content without notice.

The programs in this book are for instructional purposes only. They have been tested with care, but are not guaranteed for any particular intent beyond educational purposes. The author and the publisher do not offer any warranties or representations, nor do they accept any liabilities with respect to the programs.

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at: [www.cengage.com/global](http://www.cengage.com/global)

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

To learn more about Course Technology, visit  
[www.cengage.com/coursetechnology](http://www.cengage.com/coursetechnology)

Purchase any of our products at your local college store or at our preferred online store: [www.CengageBrain.com](http://www.CengageBrain.com)

Printed in the United States of America  
1 2 3 4 5 6 7 19 18 17 16 15 14 13

# Brief Contents

iii

Preface . . . . .	xiii
<b>CHAPTER 1</b>	A First Program Using C# . . . . .
<b>CHAPTER 2</b>	Using Data . . . . .
<b>CHAPTER 3</b>	Using GUI Objects and the Visual Studio IDE . .
<b>CHAPTER 4</b>	Making Decisions . . . . .
<b>CHAPTER 5</b>	Looping . . . . .
<b>CHAPTER 6</b>	Using Arrays . . . . .
<b>CHAPTER 7</b>	Using Methods . . . . .
<b>CHAPTER 8</b>	Advanced Method Concepts . . . . .
<b>CHAPTER 9</b>	Using Classes and Objects . . . . .
<b>CHAPTER 10</b>	Introduction to Inheritance . . . . .
<b>CHAPTER 11</b>	Exception Handling . . . . .
<b>CHAPTER 12</b>	Using Controls . . . . .
<b>CHAPTER 13</b>	Handling Events . . . . .
<b>CHAPTER 14</b>	Files and Streams . . . . .
<b>CHAPTER 15</b>	Using LINQ to Access Data in C# Programs .
<b>APPENDIX A</b>	Operator Precedence and Associativity . . .
<b>APPENDIX B</b>	Understanding Numbering Systems and Computer Codes . . . . .
<b>APPENDIX C</b>	Using The IDE Editor . . . . .
	Glossary . . . . .
	Index . . . . .

# Contents

iv

Preface . . . . .	xiii
<b>CHAPTER 1 A First Program Using C# . . . . .</b>	<b>1</b>
Programming . . . . .	2
Procedural and Object-Oriented Programming . . . . .	4
Features of Object-Oriented Programming Languages . . . . .	7
The C# Programming Language . . . . .	9
Writing a C# Program that Produces Output . . . . .	11
Selecting Identifiers . . . . .	14
Improving Programs by Adding Comments and Using the System Namespace . . . . .	17
Adding Program Comments . . . . .	17
Using the System Namespace . . . . .	19
Compiling and Executing a C# Program . . . . .	22
Compiling Code from the Command Prompt . . . . .	23
Compiling Code Using the Visual Studio IDE . . . . .	26
Noticing the Differences Between the Programs in the Text Editor and the IDE . . . . .	28
Deciding Which Environment to Use . . . . .	29
Chapter Summary . . . . .	37
Key Terms . . . . .	38
Review Questions . . . . .	41
Exercises . . . . .	44
<b>CHAPTER 2 Using Data . . . . .</b>	<b>47</b>
Declaring Variables . . . . .	48
Displaying Variable Values . . . . .	52
Using the Integral Data Types . . . . .	57
Using Floating-Point Data Types . . . . .	61
Formatting Floating-Point Values . . . . .	62
Using Arithmetic Operators . . . . .	64
Using Shortcut Arithmetic Operators . . . . .	66
Using the bool Data Type . . . . .	70
Understanding Numeric Type Conversion . . . . .	74
Using the char Data Type . . . . .	76

Using the <code>string</code> Data Type . . . . .	79
Defining Named Constants . . . . .	83
Working with Enumerations . . . . .	84
Accepting Console Input . . . . .	86
Using the <code>Convert</code> Class . . . . .	87
Using the <code>Parse()</code> Methods . . . . .	89
Chapter Summary . . . . .	91
Key Terms . . . . .	92
Review Questions . . . . .	96
Exercises . . . . .	98
<b>CHAPTER 3</b>	<b>Using GUI Objects and the Visual Studio IDE . . . 103</b>
Creating a Form in the IDE . . . . .	104
Using the Toolbox to Add a Button to a Form . . . . .	109
Adding Functionality to a Button on a Form . . . . .	112
Adding Labels and TextBoxes to a Form . . . . .	115
Understanding Focus and Tab Control . . . . .	119
Formatting Data in GUI Applications . . . . .	120
Changing a Label's Font . . . . .	120
Naming Forms and Controls . . . . .	121
Correcting Errors . . . . .	123
Deleting an Unwanted Event-Handling Method . . . . .	125
Failing to Close a Form Before Attempting to Reexecute a Program . . . . .	126
Using Visual Studio Help . . . . .	126
Deciding Which Interface to Use . . . . .	127
Chapter Summary . . . . .	135
Key Terms . . . . .	135
Review Questions . . . . .	137
Exercises . . . . .	139
<b>CHAPTER 4</b>	<b>Making Decisions . . . . . 143</b>
Understanding Logic-Planning Tools and Decision Making . . . . .	144
Making Decisions Using the <code>if</code> Statement . . . . .	147
A Note on Equivalency Comparisons . . . . .	153
Making Decisions Using the <code>if-else</code> Statement . . . . .	155
Using Compound Expressions in <code>if</code> Statements . . . . .	160
Using the Conditional AND Operator . . . . .	160
Using the Conditional OR Operator . . . . .	162
Using the Logical AND and OR Operators . . . . .	163
Combining AND and OR Operators . . . . .	163
Making Decisions Using the <code>switch</code> Statement . . . . .	168
Using an Enumeration with a <code>switch</code> Statement . . . . .	171

Using the Conditional Operator . . . . .	173
Using the NOT Operator . . . . .	175
Avoiding Common Errors When Making Decisions	
Performing Accurate and Efficient Range Checks . . . . .	177
Using <code>&amp;&amp;</code> and <code>  </code> Appropriately . . . . .	178
Using the <code>!</code> Operator Correctly . . . . .	179
Decision-Making Issues in GUI Programs . . . . .	180
Chapter Summary . . . . .	185
Key Terms . . . . .	185
Review Questions . . . . .	187
Exercises . . . . .	191
<b>CHAPTER 5      Looping . . . . .</b>	<b>197</b>
Using the <code>while</code> Loop . . . . .	198
Using the <code>for</code> Loop . . . . .	206
Using the <code>do</code> Loop . . . . .	212
Using Nested Loops . . . . .	216
Accumulating Totals . . . . .	219
Improving Loop Performance	
Avoiding Unnecessary Operations . . . . .	222
Considering the Order of Evaluation of Short-Circuit Operators .	222
Comparing to Zero . . . . .	223
Employing Loop Fusion . . . . .	225
Using Prefix Incrementing Rather than Postfix Incrementing .	225
Looping Issues in GUI Programs . . . . .	227
Chapter Summary . . . . .	230
Key Terms . . . . .	231
Review Questions . . . . .	232
Exercises . . . . .	235
<b>CHAPTER 6      Using Arrays . . . . .</b>	<b>241</b>
Declaring an Array and Assigning Values to Array Elements . . .	242
Initializing an Array . . . . .	245
Accessing Array Elements . . . . .	246
Using the <code>Length</code> Property . . . . .	247
Using <code>foreach</code> . . . . .	248
Using <code>foreach</code> with Enumerations . . . . .	249
Searching an Array Using a Loop . . . . .	251
Using a <code>for</code> Loop to Search an Array . . . . .	252
Improving a Loop's Efficiency . . . . .	254
Using a <code>while</code> Loop to Search an Array . . . . .	255
Searching an Array for a Range Match . . . . .	257

Using the <code>BinarySearch()</code> , <code>Sort()</code> , and <code>Reverse()</code> Methods . . . . .	259
Using the <code>BinarySearch()</code> Method . . . . .	259
Using the <code>Sort()</code> Method . . . . .	261
Using the <code>Reverse()</code> Method . . . . .	262
Using Multidimensional Arrays . . . . .	266
Using Jagged Arrays . . . . .	270
Array Issues in GUI Programs . . . . .	271
Chapter Summary . . . . .	274
Key Terms . . . . .	274
Review Questions . . . . .	275
Exercises . . . . .	278
<b>CHAPTER 7</b>	
<b>Using Methods . . . . .</b>	<b>283</b>
Understanding Methods and Implementation Hiding . . . . .	284
Understanding Implementation Hiding . . . . .	285
Writing Methods with No Parameters and No Return Value . . . . .	286
An Introduction to Accessibility . . . . .	287
An Introduction to the Optional <code>static</code> Modifier . . . . .	288
An Introduction to Return Types . . . . .	289
Understanding the Method Identifier . . . . .	289
Placing a Method in a Class . . . . .	290
Declaring Variables and Constants in a Method . . . . .	292
Writing Methods That Require a Single Argument . . . . .	295
Writing Methods That Require Multiple Arguments . . . . .	300
Writing a Method that Returns a Value . . . . .	301
Writing a Method that Returns a Boolean Value . . . . .	304
Analyzing a Built-in Method . . . . .	305
Passing Array Values to a Method . . . . .	308
Passing a Single Array Element to a Method . . . . .	308
Passing an Array to a Method . . . . .	309
Alternate Ways to Write a <code>Main()</code> Method Header . . . . .	312
Writing a <code>Main()</code> Method with a Parameter List . . . . .	312
Writing a <code>Main()</code> Method with an Integer Return Type . . . . .	314
Writing a <code>Main()</code> Method with <code>public</code> Access . . . . .	314
Issues Using Methods in GUI Programs . . . . .	315
Understanding Methods that are Automatically Generated in the Visual Environment . . . . .	315
Appreciating Scope in a GUI Program . . . . .	316
Creating Methods to be Nonstatic when Associated with a Form . .	316
Chapter Summary . . . . .	317
Key Terms . . . . .	318
Review Questions . . . . .	319
Exercises . . . . .	323

<b>CHAPTER 8</b>	<b>Advanced Method Concepts . . . . .</b>	<b>327</b>
	Understanding Parameter Types . . . . .	328
	Using Mandatory Value Parameters . . . . .	328
	Using Reference Parameters, Output Parameters, and Parameter Arrays . . . . .	330
	Using a <code>ref</code> Parameter . . . . .	331
	Using an <code>out</code> Parameter . . . . .	332
	Using the <code>TryParse()</code> Methods . . . . .	333
	Using Parameter Arrays . . . . .	336
	Overloading Methods . . . . .	340
	Understanding Overload Resolution . . . . .	345
	Discovering Built-In Overloaded Methods . . . . .	346
	Avoiding Ambiguous Methods . . . . .	349
	Using Optional Parameters . . . . .	351
	Leaving Out Unnamed Arguments . . . . .	353
	Using Named Arguments . . . . .	354
	Advantages to Using Named Arguments . . . . .	355
	Disadvantages to Using Named Arguments . . . . .	356
	Overload Resolution with Named and Optional Arguments . . . . .	359
	Chapter Summary . . . . .	360
	Key Terms . . . . .	360
	Review Questions . . . . .	361
	Exercises . . . . .	365
<b>CHAPTER 9</b>	<b>Using Classes and Objects . . . . .</b>	<b>369</b>
	Understanding Class Concepts . . . . .	370
	Creating a Class from Which Objects Can Be Instantiated . . . . .	372
	Creating Instance Variables and Methods . . . . .	373
	Creating Objects . . . . .	375
	Passing Objects to Methods . . . . .	378
	Creating Properties . . . . .	379
	Using Auto-Implemented Properties . . . . .	383
	More About <code>public</code> and <code>private</code> Access Modifiers . . . . .	389
	Understanding the <code>this</code> Reference . . . . .	392
	Understanding Constructors . . . . .	396
	Passing Parameters to Constructors . . . . .	397
	Overloading Constructors . . . . .	398
	Using Constructor Initializers . . . . .	400
	Using the <code>readonly</code> Modifier in a Constructor . . . . .	402
	Using Object Initializers . . . . .	404
	Overloading Operators . . . . .	407
	Declaring an Array of Objects . . . . .	412



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Handling Mouse and Keyboard Events . . . . .	654
Handling Mouse Events . . . . .	654
Handling Keyboard Events . . . . .	657
Managing Multiple Controls . . . . .	659
Defining Focus . . . . .	659
Handling Multiple Events with a Single Handler . . . . .	660
Continuing to Learn about Controls and Events . . . . .	662
Chapter Summary . . . . .	665
Key Terms . . . . .	666
Review Questions . . . . .	667
Exercises . . . . .	669
<b>CHAPTER 14 Files and Streams . . . . .</b>	<b>673</b>
Files and the File and Directory Classes . . . . .	674
Using the File and Directory Classes . . . . .	675
Understanding File Data Organization . . . . .	680
Understanding Streams . . . . .	683
Writing and Reading a Sequential Access File . . . . .	687
Writing Data to a Sequential Access Text File . . . . .	687
Reading from a Sequential Access Text File . . . . .	690
Searching a Sequential Text File . . . . .	697
Understanding Serialization and Deserialization . . . . .	703
Chapter Summary . . . . .	718
Key Terms . . . . .	719
Review Questions . . . . .	721
Exercises . . . . .	723
<b>CHAPTER 15 Using LINQ to Access Data in C# Programs . . . . .</b>	<b>727</b>
Understanding Relational Database Fundamentals . . . . .	728
Creating Databases and Table Descriptions . . . . .	731
Identifying Primary Keys . . . . .	733
Understanding Database Structure Notation . . . . .	734
Creating SQL Queries . . . . .	735
Creating an Access Database . . . . .	737
Understanding Implicitly Typed Variables . . . . .	740
Understanding LINQ . . . . .	743
Retrieving Data from an Access Database in C# . . . . .	747
Using LINQ Queries with an Access Database Table . . . . .	755
Using LINQ Operators to Sort and Group Data . . . . .	765
Chapter Summary . . . . .	773
Key Terms . . . . .	775
Review Questions . . . . .	776
Exercises . . . . .	780

<b>APPENDIX A</b>	Operator Precedence and Associativity . . . . .	<b>785</b>
<b>APPENDIX B</b>	Understanding Numbering Systems and Computer Codes . . . . .	<b>787</b>
<b>APPENDIX C</b>	Using The IDE Editor . . . . .	<b>797</b>
	Glossary . . . . .	<b>801</b>
	Index . . . . .	<b>821</b>

# Preface

xiii

*Microsoft Visual C# 2012, Fifth edition* provides the beginning programmer with a guide to developing programs in C#. C# is a language developed by the Microsoft Corporation as part of the .NET Framework and Visual Studio platform. The .NET Framework contains a wealth of libraries for developing applications for the Windows family of operating systems.

With C#, you can build small, reusable components that are well-suited to Web-based programming applications. Although similar to Java and C++, many features of C# make it easier to learn and ideal for the beginning programmer. You can program in C# using a simple text editor and the command prompt, or you can manipulate program components using Visual Studio's sophisticated Integrated Development Environment. This book provides you with the tools to use both techniques.

This textbook assumes that you have little or no programming experience. The writing is nontechnical and emphasizes good programming practices. The examples are business examples; they do not assume mathematical background beyond high school business math. Additionally, the examples illustrate one or two major points; they do not contain so many features that you become lost following irrelevant and extraneous details. This book provides you with a solid background in good object-oriented programming techniques and introduces you to object-oriented terminology using clear, familiar language.

## Organization and Coverage

*Microsoft Visual C# 2012* presents C# programming concepts, enforcing good style, logical thinking, and the object-oriented paradigm. Chapter 1 introduces you to the language by letting you create working C# programs using both the simple command line and the Visual Studio environment. In Chapter 2 you learn about data and how to input, store, and output data in C#. Chapter 3 provides a quick start to creating GUI applications. You can take two approaches:

- You can cover Chapter 3 and learn about GUI objects so that you can create more visually interesting applications in the subsequent chapters on decision making, looping, and array manipulation. These subsequent chapters confine GUI examples to the end of the chapters, so you can postpone GUI manipulation if you want.
- You can skip Chapter 3 until learning the fundamentals of decision making, looping, and array manipulation, and until studying object-oriented concepts such as classes, objects, polymorphism, inheritance, and exception handling. Then, after Chapter 11, you can return to Chapter 3 and use the built-in GUI component classes with a deeper understanding of how they work.

In Chapters 4, 5, and 6, you learn about the classic programming structures—making decisions, looping, and manipulating arrays—and how to implement them in C#. Chapters 7 and 8 provide a thorough study of methods, including passing parameters into and out of methods and overloading them.

xiv

Chapter 9 introduces the object-oriented concepts of classes, objects, data hiding, constructors, and destructors. After completing Chapters 10 and 11, you will be thoroughly grounded in the object-oriented concepts of inheritance and exception handling, and will be able to take advantage of both features in your C# programs. Chapter 12 continues the discussion of GUI objects from Chapter 3. You will learn about controls, how to set their properties, and how to make attractive, useful, graphical, and interactive programs. Chapter 13 takes you further into the intricacies of handling events in your interactive GUI programs. In Chapter 14, you learn to save data to and retrieve data from files. In Chapter 15 you learn how to interact with databases in C# programs—an increasingly valuable skill in the information-driven business world. C# supports LINQ (Language INtegrated Query) statements, which allow you to integrate SQL-like queries into C# programs.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

# Assessment

xvii

**PROGRAMMING EXERCISES** provide opportunities to practice concepts. These exercises increase in difficulty and allow students to explore each major programming concept presented in the chapter. Additional programming exercises are available in the Instructor's Resource Kit.

## CHAPTER 2 Using Data

### Review Questions

96

1. When you use a number such as 45 in a C# program, the number is a \_\_\_\_\_.  
a. literal constant      c. literal variable  
b. figurative constant      d. figurative variable
2. A variable declaration must contain all of the following *except* a(n) \_\_\_\_\_.  
a. data type      c. assigned value  
b. identifier      d. ending semicolon
3. Which of the following is true of variable declarations?  
a. Two variables of the same type can be declared.  
b. Two variables of different types can be declared.  
c. Two variables of the same type must be declared.  
d. Two variables of the same type cannot be declared.
4. Assume that you have two variables declared as int var1 and int var2. Which of the following would display 838?  
a. Console.WriteLine("{0}{1}{2}", var1, var1, var2)  
b. Console.WriteLine("{0}{1}{0}", var1, var1, var2)  
c. Console.WriteLine("{0}{1}{2}", var2, var1, var1)  
d. Console.WriteLine("{0}{1}{0}", var2, var1, var1)
5. Assume that you have a variable declared as int X. Which of the following would display X 3X?  
a. Console.WriteLine("X{0}X", var1);  
b. Console.WriteLine("X{0,2}X", var1);  
c. Console.WriteLine("X{2,0}X", var1);  
d. Console.WriteLine("X{0}{2}", var1);
6. Assume that you have a variable declared as int var1. What is the value of var1 after the following statement is executed?  
var1 = 22 % var1;  
a. 0      c. 11  
b. 1      d. 22
7. Assume that you have a variable declared as int var1. What is the value of var1 after the following statement is executed?  
var1 = 22 / var1;  
a. 1      c. 7  
b. 7      d. 11

**REVIEW QUESTIONS** test student comprehension of the major ideas and techniques presented. Twenty questions follow each chapter.

### Exercises



### Programming Exercises

235

1. a. Write a console-based application named **SumFiveDoubles** that allows the user to enter five doubles and displays their sum.  
b. Write a console-based application named **SumDoubles** that allows the user to enter any number of doubles continuously until the user enters 999. Display the sum of the values entered, not including 999.
2. Write a console-based application named **EnterLowercaseLetters** that asks the user to type a lowercase letter from the keyboard. If the character entered is a lowercase letter, display "OK"; if it is not a lowercase letter, display an error message. The program continues until the user types '!'.  
3. Write a console-based application named **CreditScores** that continuously prompts a user for consumer credit scores until the user enters a sentinel value. A valid credit score ranges from 300 through 850. When the user enters a valid score, add it to a total; when the user enters an invalid credit score, display an error message. Before the program ends, display the average of the credit scores entered.  
4. The Anderson, Bowman, and Claxton families are running their annual garage sale. Write a console-based application named **GarageSale** that prompts the user for a family initial (A, B, or C). Either uppercase or lowercase initials are valid. While the user does not type Z, continue by prompting for the amount of a sale. Keep a running total of the amount earned by each family. After the user types Z or z for an initial, display each family's total as well as a grand total made at the garage sale.
5. a. Write a console-based application named **DisplayMultiplicationTable** that displays a table of the products of every combination of two integers from 1 through 10.  
b. Create a GUI application named **DisplayMultiplicationTableGUI** that displays the multiplication table when the user clicks a button.
6. a. Write a console-based application named **OddNums** that displays all the odd numbers from 1 through 99.  
b. Create a GUI application named **OddNumsGUI** that displays all the odd numbers from 1 through 99 when the user clicks a button.
7. a. Write a console-based application named **TableOfSquares** that displays every integer value from 1 to 20, along with its squared value.  
b. Create a GUI application named **TableOfSquaresGUI** that displays every integer from 1 to 20, along with its squared value, when the user clicks a button.

**DEBUGGING EXERCISES** are included with each chapter because examining programs critically and closely is a crucial programming skill. Students can download these exercises at [www.CengageBrain.com](http://www.CengageBrain.com) and through the CourseMate available for this text. These files are also available to instructors through [login.cengage.com](http://login.cengage.com).

**CHAPTER 2****Using Data**

- 102
15. Create an enumeration named **Month** that holds values for the months of the year, starting with **JANUARY** equal to 1. Write a program named **MonthNames** that prompts the user for a month integer. Convert the user's entry to a **Month** value and display it.
  16. Create an enumeration named **Planet** that holds the names for the eight planets in our solar system, starting with **MERCURY** and ending with **NEPTUNE**. Write a program named **Planets** that prompts the user for a numeric position, and display the name of the planet that is in the requested position.
  17. Pig Latin is a nonsense language. To create a word in pig Latin, you remove the first letter and then add the first letter and "ay" at the end of the word. For example, "dog" becomes "ogday" and "cat" becomes "atcay". Write a program named **PigLatin** that allows the user to enter a word. Output the pig Latin version.

**Debugging Exercises**

1. Each of the following files in the Chapter.02 folder of your download files has syntax and/or logical errors. In each case, determine the problem. After you correct the errors, save each file using a name preceded with **Fixed**. For example, **DebugTwo1.cs** will become
- |                        |                        |
|------------------------|------------------------|
| a. <b>DebugTwo1.cs</b> | c. <b>DebugTwo3.cs</b> |
| b. <b>DebugTwo2.cs</b> | d. <b>DebugTwo4.cs</b> |

**Case Problems**

1. In Chapter 1, you created two programs to display the motto for the Greenville Idol competition that is held each summer during the Greenville County Fair. Now, write a program named **GreenvilleRevenue** that prompts a user for the number of contestants entered in last year's competition and in this year's competition. Display all the input data. Compute and display the revenue expected for this year's competition if each contestant pays a \$25 entrance fee. Also display a statement that indicates whether this year's competition has more contestants than last year's.
2. In Chapter 1, you created two programs to display the motto for Marshall's Murals. Now, write a program named **MarshallsRevenue** that prompts a user for the number of interior and exterior murals scheduled to be painted during the next month. Compute the expected revenue for each type of mural. Interior murals cost \$500 each, and exterior murals cost \$750 each. Also display the total expected revenue and a statement that indicates whether more interior murals are scheduled than exterior ones.

**CASE PROBLEMS** provide opportunities to build more detailed programs that continue to incorporate increasing functionality throughout the book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

study guides that include page references for further review. The computer-based and Internet testing components allow students to take exams at their computers, and they save the instructor time by grading each exam automatically. These test banks are also available in Blackboard-compatible formats.

- **PowerPoint Presentations:** This text provides PowerPoint slides to accompany each chapter. Slides may be used to guide classroom presentations, to make available to students for chapter review, or to print as classroom handouts. Files are provided for every figure in the text. Instructors may use the files to customize PowerPoint slides, illustrate quizzes, or create handouts.
- **Solutions:** Solutions to “You Do It” exercises and all end-of-chapter exercises are available.

## Acknowledgments

Thank you to all of the people who make this book a success, including Dan Seiter, Development Editor; Alyssa Pratt, Senior Product Manager; and Jennifer Feltri-George, Content Project Manager. I want to acknowledge every Cengage book representative who travels the country guiding instructors in their choices of educational materials.

I am also grateful to the many reviewers who provided helpful comments and encouragement during this book’s development, including Dave Curelli, Upper Cape Cod Regional Technical School; Tony Hills, Northwest State Community College; Nancy Sanchez, Baylor University; and Sylvia Unwin, Bellevue College.

Thanks, too, to my husband, Geoff, for his constant support and encouragement. Finally, this book is dedicated to the flight of angels at Rosemore Village, including (but not limited to) Amy, Becki, Carrie, Cici, Clarissa, Dawn, Emily, Jaemie, Jean, Joan, Julie, Inke, Pam, Sherri, and Tim.

*Joyce Farrell*



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

# 1

## CHAPTER

# A First Program Using C#

(Screen shots are used with permission from Microsoft.)

In this chapter you will:

- ◎ Learn about programming
- ◎ Learn about procedural and object-oriented programming
- ◎ Learn about the features of object-oriented programming languages
- ◎ Learn about the C# programming language
- ◎ Write a C# program that produces output
- ◎ Learn how to select identifiers to use within your programs
- ◎ Improve programs by adding comments and using the System namespace
- ◎ Compile and execute a C# program using the command prompt and using Visual Studio

Programming a computer is an interesting, challenging, fun, and sometimes frustrating task. As you learn a programming language, you must be precise and careful as well as creative. Computer programmers can choose from a variety of programming languages, such as Visual Basic, Java, and C++. C# (pronounced “C Sharp”) is a newer programming language that offers a wide range of options and features. As you work through this book, you will master many of them, one step at a time. If this is your first programming experience, you will learn new ways to approach and solve problems and to think logically. If you know how to program but are new to C#, you will be impressed by its capabilities.

In this chapter, you will learn about the background of programming that led to the development of C#, and you will write and execute your first C# programs.

## Programming

A computer **program** is a set of instructions that tell a computer what to do. Programs are also called **software**; software comes in two broad categories:

- **System software** describes the programs that operate the computer. Examples include operating systems such as Microsoft Windows, Mac OSX, and Linux.
- **Application software** describes the programs that allow users to complete tasks such as creating documents, calculating paychecks, and playing games.

The physical devices that make up a computer system are called **hardware**. Internally, computer hardware is constructed from circuitry that consists of small on/off switches; the most basic circuitry-level language that computers use to control the operation of those switches is called **machine language**. Machine language is expressed as a series of 1s and 0s—1s represent switches that are on, and 0s represent switches that are off. If programmers had to write computer programs using machine language, they would have to keep track of the hundreds of thousands of 1s and 0s involved in programming any worthwhile task. Not only would writing a program be a time-consuming and difficult task, but modifying programs, understanding others’ programs, and locating errors within programs all would be cumbersome. Additionally, the number and location of switches vary from computer to computer, which means you would need to customize a machine-language program for every type of machine on which the program had to run.

Fortunately, programming has become easier because of the development of high-level programming languages. A **high-level programming language** allows you to use a limited vocabulary of reasonable keywords. **Keywords** are predefined and reserved identifiers that have special meaning in a language. In other words, high-level language programs contain words such as “read,” “write,” or “add” instead of the sequence of on/off switches that perform these tasks. High-level languages also allow you to assign reasonable names to areas of computer memory; you can use names such as `hoursWorked` or `payRate`, rather than having to remember the memory locations (switch numbers) of those values.

Each high-level language has its own **syntax**, or rules of the language. For example, to produce output, you might use the verb “print” in one language and “write” in another. All languages have a specific, limited vocabulary, along with a set of rules for using that vocabulary. Programmers use a computer program called a **compiler** to translate their high-level language statements into machine code. The compiler issues an error message each time a programmer commits a **syntax error**—that is, each time the programmer uses the language incorrectly. Subsequently, the programmer can correct the error and attempt another translation by compiling the program again. The program can be completely translated to machine language only when all syntax errors have been corrected. When you learn a computer programming language such as C#, C++, Visual Basic, or Java, you really are learning the vocabulary and syntax rules for that language.



In some languages, such as BASIC, the language translator is called an interpreter. In others, such as assembly language, it is called an assembler. The various language translators operate differently, but the ultimate goal of each is to translate the higher-level language into machine language.

In addition to learning the correct syntax for a particular language, a programmer must understand computer programming logic. The **logic** behind any program involves executing the various statements and procedures in the correct order to produce the desired results. For example, you might be able to execute perfect individual notes on a musical instrument, but if you do not execute them in the proper order (or execute a B-flat when an F-sharp was expected), no one will enjoy your performance. Similarly, you might be able to use a computer language’s syntax correctly, but be unable to obtain correct results because the program is not constructed logically. Examples of logical errors include multiplying two values when you should divide them, or attempting to calculate a paycheck before obtaining the appropriate payroll data.



Programmers call some logical errors **semantic errors**. For example, if you misspell a programming language word, you commit a syntax error, but if you use a correct word in the wrong context, you commit a semantic error, generating incorrect results.

To create a working program that accomplishes its intended tasks, you must remove all syntax and logical errors from the program. This process is called **debugging** the program.



Since the early days of computer programming, program errors have been called “bugs.” The term is often said to have originated from an actual moth that was discovered trapped in the circuitry of a computer at Harvard University in 1945. Actually, the term “bug” was in use prior to 1945 to mean trouble with any electrical apparatus; even during Thomas Edison’s life, it meant an “industrial defect.” In any case, the process of finding and correcting program errors has come to be known as debugging.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

With either approach, procedural or object-oriented, you can produce a correct paycheck, and both techniques employ reusable program modules. The major difference lies in the focus the programmer takes during the earliest planning stages of a project. Taking an **object-oriented approach** to a problem means defining the objects needed to accomplish a task and developing classes that describe the objects so that each object maintains its own data and carries out tasks when another object requests them. The object-oriented approach is said to be “natural”—it is more common to think of a world of objects and the ways they interact than to consider a world of systems, data items, and the logic required to manipulate them.



Object-oriented programming employs a large vocabulary; you can learn much of this terminology in the chapter called *Using Classes and Objects*.

Originally, object-oriented programming was used most frequently for two major types of applications:

- **Computer simulations**, which attempt to mimic real-world activities so that their processes can be improved or so that users can better understand how the real-world processes operate.
- **Graphical user interfaces**, or **GUIs** (pronounced “gooeys”), which allow users to interact with a program in a graphical environment.

Thinking about objects in these two types of applications makes sense. For example, a city might want to develop a program that simulates traffic patterns to better prevent congestion. By creating a model with objects such as cars and pedestrians that contain their own data and rules for behavior, the simulation can be set in motion. For example, each car object has a specific current speed and a procedure for changing that speed. By creating a model of city traffic using objects, a computer can create a simulation of a real city at rush hour.

Creating a GUI environment for users also is a natural use for object orientation. It is easy to think of the components a user manipulates on a computer screen, such as buttons and scroll bars, as similar to real-world objects. Each GUI object contains data—for example, a button on a screen has a specific size and color. Each object also contains behaviors—for example, each button can be clicked and reacts in a specific way when clicked. Some people consider the term *object-oriented programming* to be synonymous with GUI programming, but object-oriented programming means more. Although many GUI programs are object-oriented, one does not imply the other. Modern businesses use object-oriented design techniques when developing all sorts of business applications, whether they are GUI applications or not.

## TWO TRUTHS & A LIE

### Procedural and Object-Oriented Programming

1. Procedural programs use variables and tasks that are grouped into methods or procedures.
2. Object-oriented programming languages do not support variables or methods; instead they focus on objects.
3. Object-oriented programs were first used for simulations and GUI programs.

The false statement is #2. Object-oriented programs contain variables and methods just as procedural programs do.

## Features of Object-Oriented Programming Languages

For a language to be considered object-oriented, it must support the following features:

- Classes
- Objects
- Encapsulation and interfaces
- Inheritance
- Polymorphism

A **class** describes potential objects, including their attributes and behaviors. A class is similar to a recipe or a blueprint in that it describes what features objects will have and what they will be able to do after they are created. An object is an **instance of a class**; it is one tangible example of a class.

For example, you might create a class named `Automobile`. Some of an `Automobile`'s attributes are its make, model, year, and purchase price. All `Automobiles` possess the same attributes, but not the same values, or states, for those attributes. (Programmers also call the values of an object's attributes the **properties** of the object.) When you create specific `Automobile` objects, each object can hold unique values for the attributes, such as *Ford*, *Taurus*, 2013, and \$27,000. Similarly, a `Dog` has attributes that include its breed, name, age, and vaccination status; the attributes for a particular dog might be *Labrador retriever*, *Murphy*, 7, and *current*.

When you understand that an object belongs to a specific class, you know a lot about the object. If your friend purchases an `Automobile`, you know it has *some* model name; if your friend gets a `Dog`, you know it has *some* breed. You probably don't know the current state of the `Automobile`'s speed or of the `Dog`'s shots, but you do know that those attributes exist for the `Automobile` and `Dog` classes. Similarly, in a GUI operating environment, you expect each



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Object-oriented languages also support **polymorphism**, which is the ability to create methods that act appropriately depending on the context. That is, programs written in object-oriented languages can distinguish between methods with the same name based on the type of object that uses them. For example, you are able to “fill” both a Dog and an Automobile, but you do so by very different means. Similarly, the procedure to “fill” a ShowDog might require different food than that for a “plain” Dog. Older, non-object-oriented languages could not make such distinctions, but object-oriented languages can.



The chapters *Using Classes and Objects* and *Introduction to Inheritance* contain much more information on the features of object-oriented programs.



Watch the video *Object-Oriented Programming*.

### TWO TRUTHS & A LIE

#### Features of Object-Oriented Programming Languages

1. Object-oriented programs contain classes that describe the attributes and methods of objects.
2. Object-oriented programming languages support inheritance, which refers to the packaging of attributes and methods into logical units.
3. Object-oriented programming languages support polymorphism, which is the ability of a method to act appropriately based on the context.

The false statement is #2. Inheritance is the ability to extend classes to make more specific ones. Encapsulation refers to the packaging of attributes and methods.

## The C# Programming Language

The **C# programming language** was developed as an object-oriented and component-oriented language. It is part of Microsoft Visual Studio, a package designed for developing applications that run on Windows computers. Unlike other programming languages, C# allows every piece of data to be treated as an object and to consistently employ the principles of object-oriented programming. C# provides constructs for creating components with properties, methods, and events, making it an ideal language for modern programming, where building small, reusable components is more important than building huge, stand-alone applications. You can find Microsoft’s C# specifications at [msdn.microsoft.com](http://msdn.microsoft.com). Search for *C# specifications*.

If you have not programmed before, the differences between C# and other languages mean little to you. However, experienced programmers will appreciate the thought that was put into C# features. For example:

- C# contains a GUI interface that makes it similar to Visual Basic, but C# is considered more concise than Visual Basic.
- C# is modeled after the C++ programming language, but is considered easier to learn. Some of the most difficult features to understand in C++ have been eliminated in C#.



Some differences between C# and C++ are that pointers are not used in C# (except in a mode called unsafe, which is rarely used), object destructors and forward declarations are not needed, and using `#include` files is not necessary. Multiple inheritance, which causes many C++ programming errors, is not allowed in C#.



- C# is very similar to Java, because Java was also based on C++. However, C# is more truly object-oriented. Unlike in Java, every piece of data in C# is an object, providing all data with increased functionality.



The C# programming language was standardized in 2002 by Ecma International. You can read or download this set of standards at [www.ecma-international.org/publications/standards/Ecma-334.htm](http://www.ecma-international.org/publications/standards/Ecma-334.htm).

## TWO TRUTHS & A LIE

### The C# Programming Language

1. The C# programming language was developed as an object-oriented and component-oriented language.
2. C# contains several features that make it similar to other languages such as Java and Visual Basic.
3. C# contains many advanced features, so the C++ programming language was created as a simpler version of the language.

The `false` statement is #3. C# is modeled after the C++ programming language, but some of the most difficult features to understand in C++ have been eliminated in C#.

## Writing a C# Program that Produces Output

At first glance, even the simplest C# program involves a fair amount of confusing syntax. Consider the simple program in Figure 1-1. This program is written on seven lines, and its only task is to display “This is my first C# program” on the screen.

11

```
class FirstClass
{
    static void Main()
    {
        System.Console.WriteLine("This is my first C# program");
    }
}
```

**Figure 1-1** FirstClass console application

The statement that does the actual work in this program is in the middle of the figure:

```
System.Console.WriteLine("This is my first C# program");
```

The statement ends with a semicolon because all C# statements do.

The text “This is my first C# program” is a **literal string** of characters—that is, a series of characters that will be used exactly as entered. Any literal string in C# appears between double quotation marks.

The string “This is my first C# program” appears within parentheses because the string is an argument to a method, and arguments to methods always appear within parentheses.

**Arguments** represent information that a method needs to perform its task. For example, if making an appointment with a dentist’s office was a C# method, you would write the following:

```
MakeAppointment("September 10", "2 p.m.");
```

Accepting and processing a dental appointment is a method that consists of a set of standard procedures. However, each appointment requires different information—the date and time—and this information can be considered the arguments of the `MakeAppointment()` method. If you make an appointment for September 10 at 2 p.m., you expect different results than if you make one for September 9 at 8 a.m. or December 25 at midnight. Likewise, if you pass the argument “Happy Holidays” to a method, you will expect different results than if you pass the argument “This is my first C# program”.

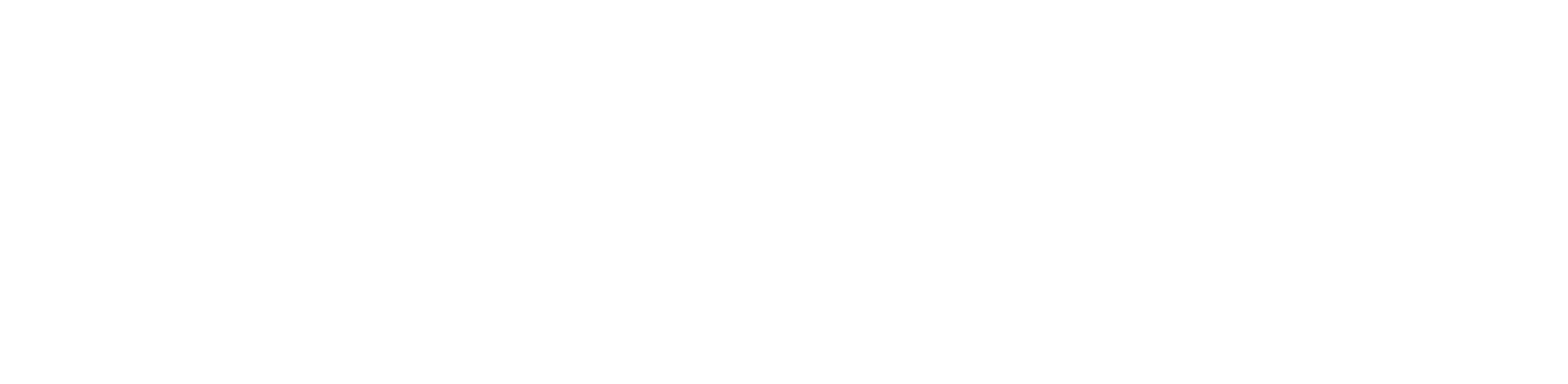


Although a string can be an argument to a method, not all arguments are strings. In this book, you will see and write methods that accept many other types of data.

Within the statement `System.Console.WriteLine("This is my first C# program");`, the method to which you are passing the argument string “This is my first C# program” is named `WriteLine()`. The **WriteLine() method** displays output on the screen and positions the



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

## Selecting Identifiers

Every method that you use within a C# program must be part of a class. To create a class, you use a class header and curly braces in much the same way you use a header and braces for a method within a class. When you write `class FirstClass`, you are defining a class named `FirstClass`. A class name does not have to contain the word *Class* as `FirstClass` does; as a matter of fact, most class names you create will not contain *Class*. You can define a C# class using any identifier you need, as long as it meets the following requirements:

- An identifier must begin with an underscore, the “at” sign (@), or a letter. (Letters include foreign-alphabet letters such as Π and Ω, which are contained in the set of characters known as Unicode. You will learn more about Unicode in the next chapter.)
- An identifier can contain only letters, digits, underscores, and the “at” sign. An identifier cannot contain spaces or any other punctuation or special characters such as #, \$, or &.
- An identifier cannot be a C# reserved keyword, such as `class` or `void`. Table 1-1 provides a complete list of reserved keywords. (Actually, you can use a keyword as an identifier if you precede it with an “at” sign, as in `@class`. An identifier with an @ prefix is a **verbatim identifier**. This feature allows you to use code written in other languages that do not have the same set of reserved keywords. However, when you write original C# programs, you should not use the keywords as identifiers.)

---

abstract	float	return
as	for	sbyte
base	foreach	sealed
bool	goto	short
break	if	sizeof
byte	implicit	stackalloc
case	in	static
catch	int	string
char	interface	struct
checked	internal	switch
class	is	this
const	lock	throw
continue	long	true
decimal	namespace	try
default	new	typeof

---

**Table 1-1** C# reserved keywords (continues)

(continued)

delegate	null	uint
do	object	ulong
double	operator	unchecked
else	out	unsafe
enum	override	ushort
event	params	using
explicit	private	virtual
extern	protected	void
false	public	volatile
finally	readonly	while
fixed	ref	

15

**Table 1-1** C# reserved keywords

The following identifiers have special meaning in C# but are not keywords: `add`, `alias`, `get`, `global`, `partial`, `remove`, `set`, `value`, `where`, and `yield`. For clarity, you should avoid using these words as your own identifiers.

Table 1-2 lists some valid and conventional class names you might use when creating classes in C#. You should follow established conventions for C# so that other programmers can interpret and follow your programs. Table 1-3 lists some class names that are valid, but unconventional; Table 1-4 lists some illegal class names.

Class Name	Description
<code>Employee</code>	Begins with an uppercase letter
<code>FirstClass</code>	Begins with an uppercase letter, contains no spaces, and has an initial uppercase letter that indicates the start of the second word
<code>PushButtonControl</code>	Begins with an uppercase letter, contains no spaces, and has an initial uppercase letter that indicates the start of all subsequent words
<code>Budget2012</code>	Begins with an uppercase letter and contains no spaces

**Table 1-2** Some valid and conventional class names in C#

Class Name	Description
employee	Unconventional as a class name because it begins with a lowercase letter
First_Class	Although legal, the underscore is not commonly used to indicate new words in class names
Pushbuttoncontrol	No uppercase characters are used to indicate the start of a new word, making the name difficult to read
BUDGET2013	Unconventional as a class name because it contains all uppercase letters
Void	Although this identifier is legal because it is different from the keyword void, which begins with a lowercase v, the similarity could cause confusion

**Table 1-3** Some unconventional (though legal) class names in C#

Class Name	Description
an employee	Space character is illegal
Push Button Control	Space characters are illegal
class	class is a reserved word
2011Budget	Class names cannot begin with a digit
phone#	The # symbol is not allowed; identifiers consist of letters, digits, underscores, or @

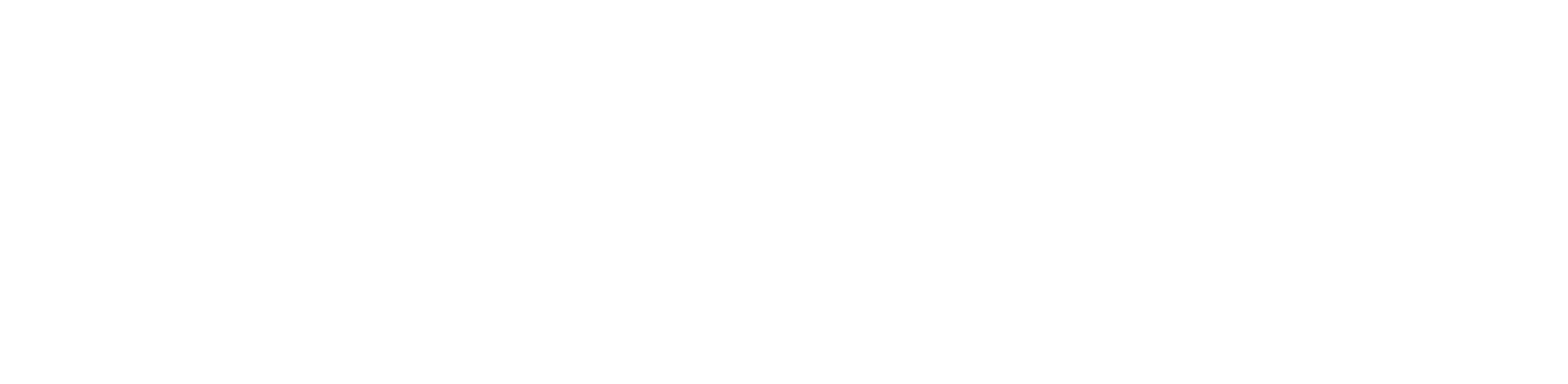
**Table 1-4** Some illegal class names in C#

In Figure 1-1, the line `class FirstClass` contains the keyword `class`, which identifies `FirstClass` as a class.

The simple program shown in Figure 1-1 has many pieces to remember. For now, you can use the program shown in Figure 1-2 as a shell, where you replace the identifier `AnyLegalClassName` with any legal class name, and the line `*****` with any statements that you want to execute.

```
class AnyLegalClassName
{
    static void Main()
    {
        *****/;
    }
}
```

**Figure 1-2** Shell program



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



As you work through this book, you should add comments as the first few lines of every program file. The comments should contain your name, the date, and the name of the program. Your instructor might want you to include additional comments.

18

Comments also can be useful when you are developing a program. If a program is not performing as expected, you can **comment out** various statements and subsequently run the program to observe the effect. When you comment out a statement, you turn it into a comment so that the compiler will ignore it. This approach helps you pinpoint the location of errant statements in malfunctioning programs.

C# offers three types of comments:

- **Line comments** start with two forward slashes (//) and continue to the end of the current line. Line comments can appear on a line by themselves, or they can occupy part of a line following executable code.
- **Block comments** start with a forward slash and an asterisk /\*) and end with an asterisk and a forward slash (\*). Block comments can appear on a line by themselves, on a line before executable code, or after executable code. When a comment is long, block comments can extend across as many lines as needed.
- C# also supports a special type of comment used to create documentation within a program. These comments, called **XML-documentation format comments**, use a special set of tags within angle brackets (< >). (XML stands for Extensible Markup Language.) You will learn more about this type of comment as you continue your study of C#.



The forward slash (/) and the backslash (\) characters often are confused, but they are distinct characters. You cannot use them interchangeably.

Figure 1-3 shows how comments can be used in code. The program covers 10 lines, yet only seven are part of the executable C# program, including the last two lines, which contain curly braces and are followed by partial-line comments. The only line that actually *does* anything visible when the program runs is the shaded one that displays “Message”.

```
/* This program is written to demonstrate
   using comments */
class ClassWithOneExecutingLine
{
    static void Main()
    {
        // The next line writes the message
        System.Console.WriteLine("Message");
    } // End of Main
} // End of ClassWithOneExecutingLine
```

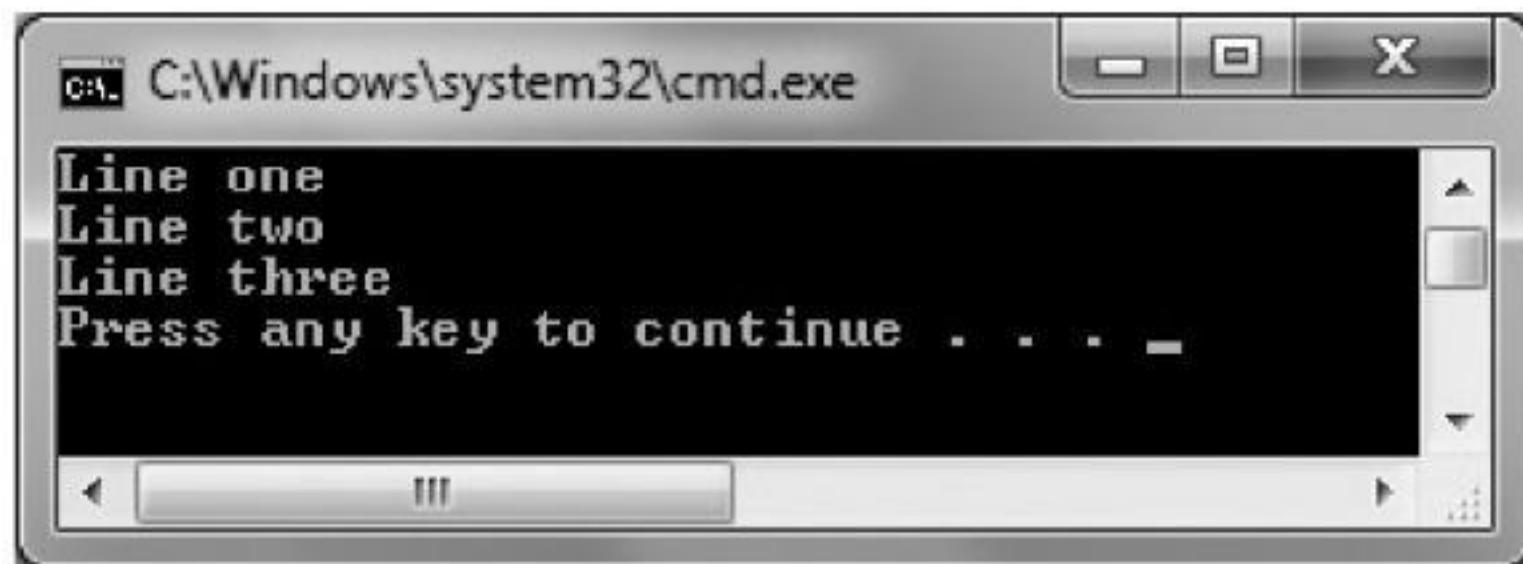
**Figure 1-3** Using comments within a program

## Using the System Namespace

A program can contain as many statements as you need. For example, the program in Figure 1-4 produces the three lines of output shown in Figure 1-5. (To get the output, you have to know how to compile and execute the program, which you will learn in the next part of this chapter.) A semicolon separates each program statement.

```
class ThreeLinesOutput
{
    static void Main()
    {
        System.Console.WriteLine("Line one");
        System.Console.WriteLine("Line two");
        System.Console.WriteLine("Line three");
    }
}
```

**Figure 1-4** A program that produces three lines of output



**Figure 1-5** Output of ThreeLinesOutput program



Figure 1-5 shows the output of the `ThreeLinesOutput` program when it is run in Visual Studio. The prompt to press any key to continue is not part of the program; it is added by Visual Studio, but it does not appear if you run the program from the command prompt.

The program in Figure 1-4 shows a lot of repeated code—the phrase `System.Console.WriteLine` appears three times. When you need to repeatedly use a class from the same namespace, you can shorten the statements you type by adding a clause that indicates a namespace containing the class. You indicate a namespace with a **using clause**, or **using directive**, as shown in the shaded statement in the program in Figure 1-6. If you type `using System;` prior to the class definition, the compiler knows to use the `System` namespace when it encounters the `Console` class. The output of the program in Figure 1-6 is identical to that in Figure 1-4, in which `System` was repeated with each `WriteLine()` statement.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

(continued)

any language produces “Hello, world!” as its output. To create a C# program, you can use any simple text editor, such as Notepad, or the editor that is included as part of Microsoft Visual Studio. There are advantages to using the C# editor to write your programs, but using a plain text editor is simpler when you are getting started.

### *Entering a Program into an Editor*

1. Start any text editor, such as Notepad, and open a new document, if necessary.
2. Type the `using` statement and the header for the class:

```
using System;
class Hello
```

3. On the next two lines, type the class-opening and class-closing curly braces: `{}`. Many programmers type a closing brace as soon as they type the opening one to guarantee that they always type complete pairs.
4. Between the class braces, insert a new line, type three spaces to indent, and write the `Main()` method header:  
  
`static void Main()`
5. On the next two lines, type the opening and closing braces for the `Main()` method, indenting them about three spaces.
6. Between the `Main()` method’s braces, insert a new line and type six spaces so the next statement will be indented within the braces. Type the one executing statement in this program:

```
Console.WriteLine("Hello, world!");
```

Your code should look like Figure 1-7.

```
using System;
class Hello
{
    static void Main()
    {
        Console.WriteLine("Hello, world!");
    }
}
```

**Figure 1-7** The Hello class

(continued)

(continued)

7. Choose a location that is meaningful to you to save your program. For example, you might create a folder named C# on your hard drive. Within that folder, you might create a folder named Chapter.01 in which you will store all the examples and exercises in this chapter. If you are working in a school lab, you might be assigned a storage location on your school's server, or you might prefer to store your examples on a USB drive or other portable storage media. Save the program as **Hello.cs**. It is important that the file extension be .cs, which stands for C Sharp. If the file has a different extension, the compiler for C# will not recognize the program as a C# program.



Many text editors attach their own filename extension (such as .txt or .doc) to a saved file. Double-check your saved file to ensure that it does not have a double extension (as in Hello.cs.txt). If the file has a double extension, rename it. If you use a word-processing program as your editor, select the option to save the file as a plain text file.

## Compiling and Executing a C# Program

After you write and save a program, two more steps must be performed before you can view the program output:

1. You must compile the program you wrote (called the **source code**) into **intermediate language (IL)**.
2. The C# **just in time (JIT)** compiler must translate the intermediate code into executable code.

When you compile a C# program, your source code is translated into intermediate language. The JIT compiler converts IL instructions into native code at the last moment, and appropriately for each type of operating system on which the code might eventually be executed. In other words, the same set of IL can be JIT-compiled and executed on any supported architecture.



Some developers say that languages like C# are “semi-compiled.” That is, instead of being translated immediately from source code to their final executable versions, programs are compiled into an intermediate version that is later translated into the correct executable statements for the environment in which the program is running.

You can write a program using a simple editor such as Notepad and then perform these steps from the command line in your system. You also can write a program within the Integrated Development Environment that comes with Visual Studio. Both methods can produce the same output; the one you use is a matter of preference.

- The **command line** is the line on which you type a command in a system that uses a text interface. The **command prompt** is a request for input that appears at the beginning of the command line. In DOS, the command prompt indicates the disk drive and optional path, and ends with >. You might prefer the simplicity of the command line because you do not work with multiple menus and views. Additionally, if you want to pass command-line arguments to a program, you must compile from the command line.
- The **Integrated Development Environment (IDE)** is a programming environment that allows you to issue commands by selecting choices from menus and clicking buttons. Many programmers prefer using the IDE because it provides features such as color-coded keywords and automatic statement completion.

## Compiling Code from the Command Prompt



If you will be using the IDE to write all your programs, you can read this section quickly, and then concentrate on the section titled “Compiling Code Using the Visual Studio IDE.”

To compile your source code from the command line, you first locate the command prompt. There, you type `csc`, followed by the name of the file that contains the source code. The command `csc` stands for “C Sharp compiler.” For example, to compile a file named `ThreeLinesOutput.cs`, you would type the following and then press the Enter key:

```
csc ThreeLinesOutput.cs
```

One of three outcomes will occur:

- You receive an operating system error message such as “Bad command or file name” or “`csc` is not recognized as an internal or external command, operable program or batch file”. You can recognize operating system messages because they do not start with the name of the program you are trying to compile.
- You receive one or more program language error messages. You can recognize program language error messages because they start with the name of the program followed by a line number and the position where the error was first noticed.
- You receive no error messages (only a copyright statement from Microsoft), indicating that the program has compiled successfully.

### *What to Do if You Receive an Operating System Error Message at the Command Prompt*

If you receive an operating system message such as “`csc` is not recognized...,” or “Source file... could not be found,” it may mean that:

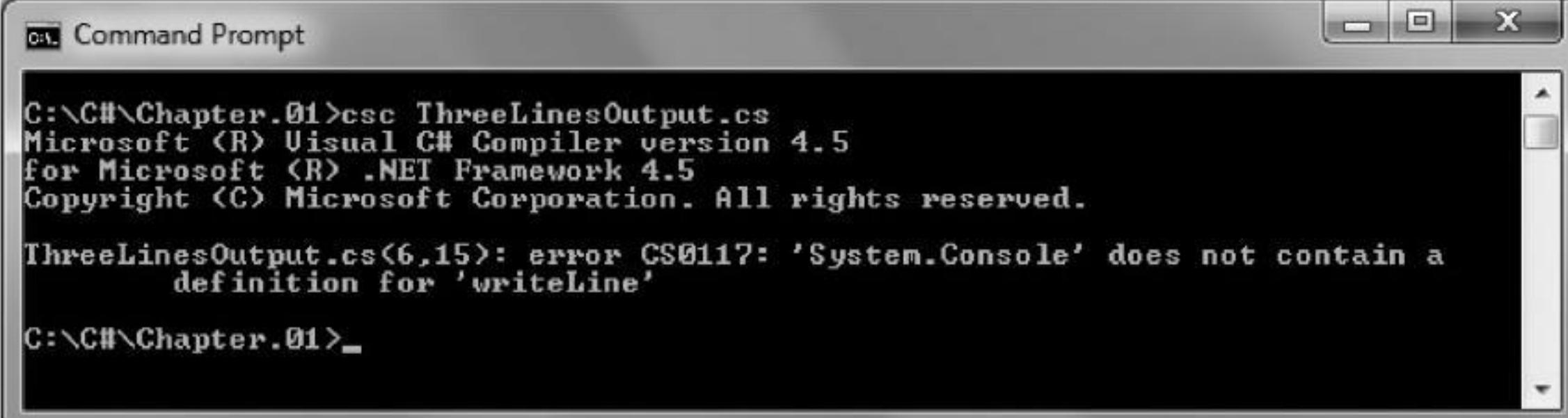
- You misspelled the command `csc`.
- You misspelled the filename.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

## What to Do if You Receive a Programming Language Error Message at the Command Prompt

If you receive a programming language error message, it means that the compiler was installed correctly, but that the source code contains one or more syntax errors. A syntax error occurs when you introduce typing errors into your program. Program error messages start with the program name, followed by parentheses that hold the line number and the position in the line where the compiler noticed the error. For example, in Figure 1-9, an error is found in ThreeLinesOutput.cs in line 6, position 15. In this case, the message is generated because `WriteLine` is typed as `writeLine` (with a lowercase `w`). The error message is “`System.Console`’ does not contain a definition for ‘`writeLine`’”. If a problem like this occurs, you must reopen the text file that contains the source code, make the necessary corrections, save the file, and compile it again.



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The command entered is "C:\C#\Chapter.01>csc ThreeLinesOutput.cs". The output shows the Microsoft C# Compiler version 4.5 and a copyright notice. Below that, an error message is displayed: "ThreeLinesOutput.cs(6,15): error CS0117: 'System.Console' does not contain a definition for 'writeLine'". The prompt at the bottom is "C:\C#\Chapter.01>\_".

**Figure 1-9** Error message generated when `WriteLine` is mistyped in the `ThreeLinesOutput` program compiled from the command prompt



The C# compiler issues warnings as well as errors. A warning is less serious than an error; it means that the compiler has determined you have done something unusual, but not illegal. If you have purposely introduced a warning situation to test a program, then you can ignore the warning. Usually, however, you should treat a warning message just as you would an error message and attempt to remedy the situation.

## What to Do When the Program Compiles Successfully at the Command Prompt

If you receive no error messages after compiling the code, then the program compiled successfully, and a file with the same name as the source code—but with an .exe extension—is created and saved in the same folder as the program text file. For example, if `ThreeLinesOutput.cs` compiles successfully, then a file named `ThreeLinesOutput.exe` is created.

To run the program from the command line, you simply type the program name—for example, `ThreeLinesOutput`. You can also type the full filename, `ThreeLinesOutput.exe`, but it is not necessary to include the extension.

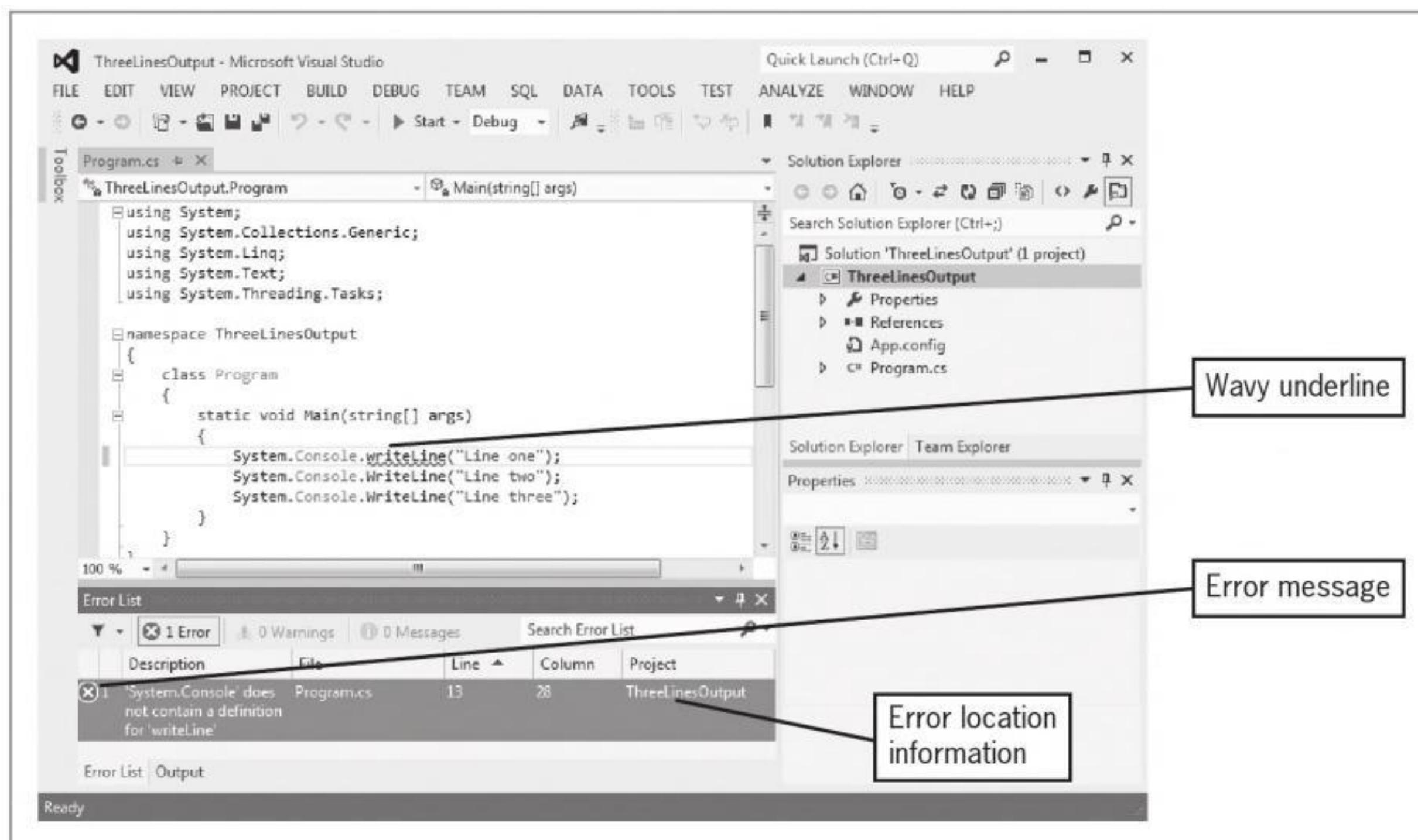


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

One way to compile a program from Visual Studio is to select BUILD from the main menu and then select Build Solution. As an alternative, you can press the F7 key. You can also select DEBUG from the main menu, and then Start Without Debugging. The advantage of the latter option is that the program will be compiled if there are no syntax errors and then execute so that you can see the output.

### *What to Do if You Receive a Programming Language Error Message in the IDE*

If you introduce a syntax error into a program in the IDE, you receive a programming language error message. For example, in Figure 1-11, `WriteLine` is spelled incorrectly because it uses a lowercase `w`. The error message displayed at the bottom of the IDE is the same one displayed for an application run at the command prompt: “`System.Console`’ does not contain a definition for ‘`writeLine`’”. The position is given, and as an additional visual aid, a wavy underline emphasizes the unrecognized definition. When you fix the problem and compile again, the error message is eliminated.



**Figure 1-11** Error message generated when `WriteLine` is mistyped in the `ThreeLinesOutput` program compiled in the IDE

### *What to Do When the Program Compiles Successfully in the IDE*

If you receive no error messages after compiling the code, then the program compiled successfully, and you can run the program. Select DEBUG from the main menu, and then select Start Without Debugging. The output appears, as you saw it in Figure 1-5 earlier in this chapter.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



## You Do It

### Compiling and Executing a Program from the Command Line



If you do not plan to use the command line to execute programs, you can skip to the next part of this “You Do It” section, called “Compiling and Executing a Program Using the Visual Studio IDE.”

1. Go to the command prompt on your system. For example, in Windows 7, click **Start**, then click **All Programs**, click **Accessories**, and click **Command Prompt**. In Windows 8, click the **Windows Explorer** icon on the **Start** screen. In the **Libraries** window that opens, click the **File** menu, and click **Open command prompt**. In the next window, click **Open command prompt** again. Change the current directory to the name of the folder that holds your program.

If your command prompt indicates a path other than the one you want, you can type `cd\` and then press Enter to return to the root directory. You can then type `cd` to change the path to the one where your program resides. For example, if you stored your program file in a folder named `Chapter.01` within a folder named `C#`, then you can type the following:

`cd C#\Chapter.01`

The command `cd` is short for *change directory*.

2. Type the command that compiles your program:

`csc Hello.cs`

If you receive no error messages and the prompt returns, it means that the compile operation was successful, that a file named `Hello.exe` has been created, and that you can execute the program. If you do receive error messages, check every character of the program you typed to make sure it matches Figure 1-7 in the last “You Do It” section. Remember, C# is case sensitive, so all casing must match exactly. When you have corrected the errors, repeat this step to compile the program again.

3. You can verify that a file named `Hello.exe` was created in several ways:

- At the command prompt, type `dir` to view a directory of the files stored in the current folder. Both `Hello.cs` and `Hello.exe` should appear in the list.
- Use Windows Explorer to view the contents of the `Chapter.01` folder, verifying that two `Hello` files are listed.

(continues)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

(continued)



35

**Figure 1-16** Output of the Hello application in Visual Studio

8. Close Visual Studio by clicking **FILE** on the menu bar and then clicking **Exit**, or by clicking the **Close** box in the upper-right corner of the Visual Studio window.
9. When you create a C# program using an editor such as Notepad and compiling with the `csc` command, only two files are created—`Hello.cs` and `Hello.exe`. When you create a C# program using the Visual Studio editor, many additional files are created. You can view their filenames in several ways:
  - At the command prompt, type **dir** to view a directory of the files stored in the folder where you saved the project (for example, your `Chapter.01` folder). Within the folder, a new folder named `Hello` has been created. Type the command **cd Hello** to change the current path to include this new folder, then type **dir** again. You see another folder named `Hello`. Type **cd Hello** again, and **dir** again. You should see several folders and files.
  - Double-click the **Computer** icon, find and double-click the correct drive, select the **C#** folder and the **Chapter.01** folder (or the path you are using), double-click the **Hello** folder, and view the contents. Double-click the second **Hello** folder and view the contents there too.
  - Use Windows Explorer to view the contents of the `Hello` folders within the **Chapter.01** folder.

Regardless of the technique you use to examine the folders, you will find that the innermost `Hello` folder contains a `bin` folder, an `obj` folder, a `Properties` folder, and additional files. If you explore further, you will find that the `bin` folder contains `Debug` and `Release` folders, which include additional files. Using the Visual Studio editor to compile your programs creates a significant amount of overhead. These additional files become important as you create more sophisticated C# projects.

(continues)

(continued)



Depending on the version of Visual Studio you are using, your folder configuration might be slightly different.



If you followed the earlier instructions on compiling a program from the command line, and you used the same folder when using the IDE, you will see the additional Hello.cs and Hello.exe files in your folder. These files will have an earlier time stamp than the files you just created. If you were to execute a new program within Visual Studio without saving and executing it from the command line first, you would not see these two additional files.

### *Adding Comments to a Program*

Comments are nonexecuting statements that help to document a program. In the following steps, you add some comments to the program you just created.

1. If you prefer compiling programs from the command line, then open the **Hello.cs** file in your text editor. If you prefer compiling programs from Visual Studio, then open Visual Studio, click **FILE**, point to **Open**, click **Project/Solution**, browse for the correct folder, double-click the **Hello** folder, and then double-click the **Hello** file.
2. Position your cursor at the top of the file, press **Enter** to insert a new line, press the **Up** arrow key to go to that line, and then type the following comments at the top of the file. Press **Enter** after typing each line. Insert your name and today's date where indicated.

```
//Filename Hello.cs  
//Written by <your name>  
//Written on <today's date>
```

3. Scroll to the line that reads `static void Main()` and press **Enter** to start a new line. Then press the **Up** arrow; in the new blank line, aligned with the start of the `Main()` method header, type the following block comment in the program:

```
/* This program demonstrates the use of  
the WriteLine() method to display the  
message Hello, world! */
```

4. Save the file, replacing the old Hello.cs file with this new, commented version.

(continues)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- You can improve programs by adding comments, which are nonexecuting statements that you add to document a program or to disable statements when you test a program. The three types of comments in C# are line comments that start with two forward slashes (//) and continue to the end of the current line, block comments that start with a forward slash and an asterisk /\*) and end with an asterisk and a forward slash (\*), and XML-documentation comments. You can also improve programs and shorten the statements you type by using a clause that indicates a namespace where your classes can be found.
- To create a C# program, you can use the Microsoft Visual Studio environment or any text editor, such as Notepad. After you write and save a program, you must compile the source code into intermediate and machine language.

## Key Terms

A computer **program** is a set of instructions that tell a computer what to do.

**Software** is computer programs.

**System software** describes the programs that operate the computer.

**Application software** is the programs that allow users to complete tasks.

**Hardware** comprises all the physical devices associated with a computer.

**Machine language** is the most basic circuitry-level language.

A **high-level programming language** allows you to use a vocabulary of keywords instead of the sequence of on/off switches that perform these tasks.

**Keywords** are predefined and reserved identifiers that have special meaning to the compiler.

A language's **syntax** is its set of rules.

A **compiler** is a computer program that translates high-level language statements into machine code.

A **syntax error** is an error that occurs when a programming language is used incorrectly.

The **logic** behind any program involves executing the various statements and methods in the correct order to produce the desired results.

**Semantic errors** are the type of logical errors that occur when you use a correct word in the wrong context, generating incorrect results.

**Debugging** a program is the process of removing all syntax and logical errors from the program.

A **procedural program** is created by writing a series of steps or operations to manipulate values.

**Variables** are named computer memory locations that hold values that might vary.

An **identifier** is the name of a program component such as a variable, class, or method.

**Camel casing**, also called **lower camel casing**, is a style of creating identifiers in which the first letter is not capitalized, but each new word is.

**Pascal casing**, also called **upper camel casing**, is a style of creating identifiers in which the first letter of all new words in a name, even the first one, is capitalized.

**Methods** are compartmentalized, named program units containing instructions that accomplish tasks.

A program **calls** or **invokes** methods.

**Object-oriented programming (OOP)** is a programming technique that features objects, classes, encapsulation, interfaces, polymorphism, and inheritance.

An **object** is a concrete entity that has attributes and behaviors; an object is an instance of a class.

The **attributes of an object** represent its characteristics.

The **state of an object** is the collective value of all its attributes at any point in time.

The **behaviors of an object** are its methods.

Taking an **object-oriented approach** to a problem means defining the objects needed to accomplish a task and developing classes that describe the objects so that each maintains its own data and carries out tasks when another object requests them.

**Computer simulations** are programs that attempt to mimic real-world activities to foster a better understanding of them.

**Graphical user interfaces**, or **GUIs** (pronounced “gooeys”), are program elements that allow users to interact with a program in a graphical environment.

A **class** is a category of objects or a type of object.

An **instance of a class** is an object.

The **properties** of an object are its values.

**Encapsulation** is the technique of packaging an object's attributes and methods into a cohesive unit that can be used as an undivided entity.

A **black box** is a device you use without regard for the internal mechanisms.

An **interface** is the interaction between a method and an object.

**Inheritance** is the ability to extend a class so as to create a more specific class that contains all the attributes and methods of a more general class; the extended class usually contains new attributes or methods as well.

**Polymorphism** is the ability to create methods that act appropriately depending on the context.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

**Block comments** start with a forward slash and an asterisk /\*) and end with an asterisk and a forward slash (\*/). Block comments can appear on a line by themselves, on a line before executable code, or after executable code. They can also extend across as many lines as needed.

**XML-documentation format comments** use a special set of tags within angle brackets to create documentation within a program.

41

A **using clause** or **using directive** declares a namespace.

**Source code** is the statements you write when you create a program.

**Intermediate language (IL)** is the language into which source code statements are compiled.

The C# **just in time (JIT)** compiler translates intermediate code into executable code.

The **command line** is the line on which you type a command in a system that uses a text interface.

The **command prompt** is a request for input that appears at the beginning of the command line.

An **Integrated Development Environment (IDE)** is a program development environment that allows you to select options from menus or by clicking buttons. An IDE provides such helpful features as color coding and automatic statement completion.

## **Review Questions**



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.





You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

8. Write a program named **Initials** that displays your initials in a pattern on the screen. Compose each initial with six lines of smaller initials, as in the following example:

```
J      FFFFFF  
J      F  
J      FFF  
J      F  
J  J    F  
JJJJJJ  F
```

9. From 1925 through 1963, Burma Shave advertising signs appeared next to highways all across the United States. There were always four or five signs in a row containing pieces of a rhyme, followed by a final sign that read “Burma Shave.” For example, one set of signs that has been preserved by the Smithsonian Institution reads as follows:

```
Shaving brushes  
You'll soon see 'em  
On a shelf  
In some museum  
Burma Shave
```

Find a classic Burma Shave rhyme on the Web and write a program named **BurmaShave** that displays the rhyme.



## Debugging Exercises

1. Each of the following files in the Chapter.01 folder of your downloadable student files has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with *Fixed*. For example, DebugOne1.cs will become FixedDebugOne1.cs.
  - a. DebugOne1.cs
  - b. DebugOne2.cs
  - c. DebugOne3.cs
  - d. DebugOne4.cs



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

# CHAPTER

# 2

# Using Data

(Screen shots are used with permission from Microsoft.)

In this chapter you will:

- ◎ Learn about declaring variables
- ◎ Display variable values
- ◎ Learn about the integral data types
- ◎ Learn about floating-point data types
- ◎ Use arithmetic operators
- ◎ Learn about the `bool` data type
- ◎ Learn about numeric type conversion
- ◎ Learn about the `char` data type
- ◎ Learn about the `string` data type
- ◎ Define named constants and enumerations
- ◎ Accept console input



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

if they contain multiple words. As with class names, a variable name cannot be a C# keyword. Table 1-1 in Chapter 1 lists all the C# keywords.

You must declare all variables you want to use in a program. A **variable declaration** is the statement that names a variable and reserves storage for it. The declaration includes:

- 50
- The data type that the variable will store
  - The variable's name (its identifier)
  - An optional assignment operator and assigned value when you want a variable to contain an initial value
  - An ending semicolon

For example, the following statement declares a variable of type `int` named `myAge` and assigns it an initial value of 25:

```
int myAge = 25;
```

In other words, the statement reserves four bytes of memory with the name `myAge`, and the value 25 is stored there. The declaration is a complete statement that ends in a semicolon.

The equal sign (=) is the **assignment operator**; any value to the right of the assignment operator is assigned to the identifier to the left. An assignment made when a variable is declared is an **initialization**; an assignment made later is simply an **assignment**. Thus, the following statement initializes `myAge` to 25:

```
int myAge = 25;
```

A statement such as the following assigns a new value to the variable:

```
myAge = 42;
```

Notice that the data type is not used again when an assignment is made; it is used only in a declaration.

Also note that the expression `25 = myAge`; is illegal because assignment always takes place from right to left. By definition, a constant cannot be altered, so it is illegal to place one (such as 25) on the left side of an assignment operator. The assignment operator means “is assigned the value of the following expression.” In other words, the statement `myAge = 25` can be read as “`myAge` is assigned the value of the following expression: 25.”

Instead of using a type name from the Type column of Table 2-1, you can use the fully qualified type name from the System namespace that is listed in the System Type column. For example, instead of using the type name `int`, you can use the full name `System.Int32`. The number 32 in the name `System.Int32` represents the number of bits of storage allowed for the data type. There are 8 bits in a byte, and an `int` occupies 4 bytes. It’s better to use the shorter alias `int`, however, for the following reasons:

- The shorter alias is easier to type and read.
- The shorter alias resembles type names used in other languages such as Java and C++.
- Other C# programmers expect the shorter type names.

The variable declaration `int myAge;` declares a variable of type `int` named `myAge`, but no value is assigned at the time of creation. You can make an assignment later in the program, but you cannot use the variable in a calculation or display the value of the variable until you assign a value to it.

You can declare multiple variables of the same type in separate statements. For example, the following statements declare two variables. The first variable is named `myAge` and its value is 25. The second variable is named `yourAge` and its value is 19.

```
int myAge = 25;  
int yourAge = 19;
```

You also can declare multiple variables of the same type in a single statement by using the type once and separating the variable declarations with a comma, as shown in the following statement:

```
int myAge = 25, yourAge = 19;
```

Some programmers prefer to use the data type once and break the declaration across multiple lines, as in the following example:

```
int myAge = 25,  
    yourAge = 19;
```

When a statement occupies more than one line, it is easier to read if lines after the first one are indented a few spaces. This book follows that convention.

When you declare multiple variables of the same type, a comma separates the variable names and a single semicolon appears at the end of the declaration statement, no matter how many lines the declaration occupies. However, when declaring variables of different types, you must use a separate statement for each type. The following statements declare two variables of type `int` (`myAge` and `yourAge`) and two variables of type `double` (`mySalary` and `yourSalary`), without assigning initial values to any of them:

```
int myAge, yourAge;  
double mySalary, yourSalary;
```

Similarly, the following statements declare two `ints` and two `doubles`, assigning values to two of the four named variables:

```
int    numCarsIOwn = 2,  
        numCarsYouOwn;  
double myCarsMpg,  
      yourCarsMpg = 31.5;
```



Watch the video *Declaring Variables*.

## TWO TRUTHS & A LIE

### Declaring Variables

1. A constant value cannot be changed after a program is compiled, but a variable can be changed.
2. A data type describes the format and size of a data item and the types of operations that can be performed with it.
3. A variable declaration requires a data type, name, and assigned value.

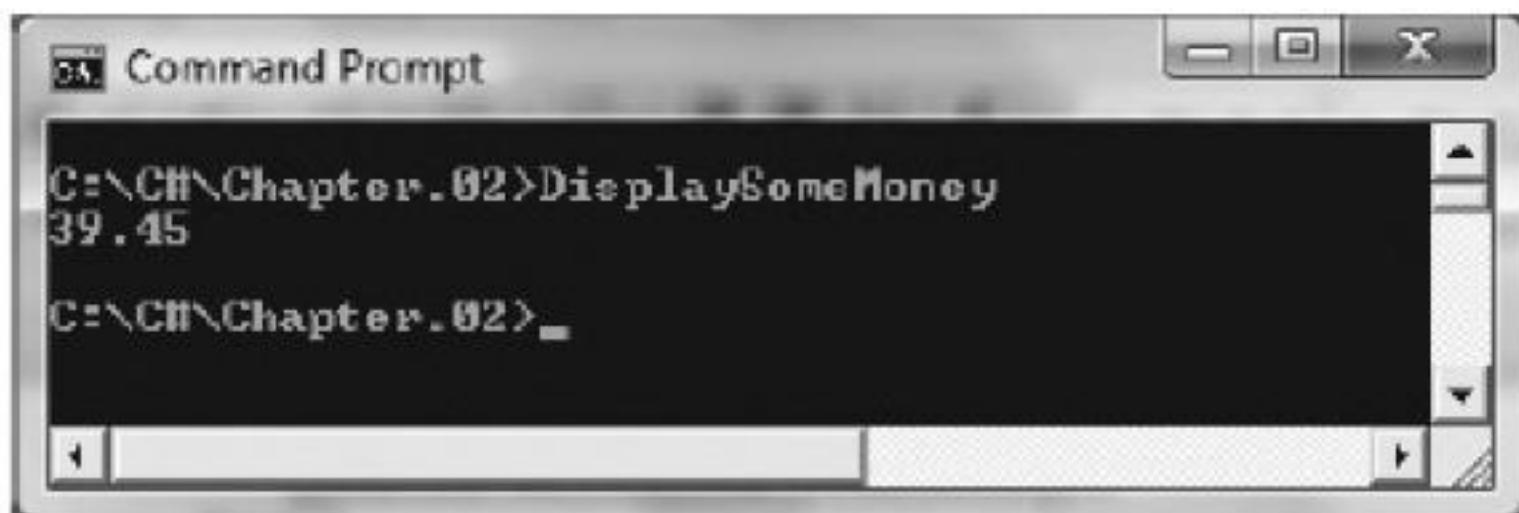
The false statement is #3. A variable declaration names a variable and reserves storage for it; it includes the data type that the variable will store, and an identifier. An assignment operator and assigned value can be included, but they are not required.

## Displaying Variable Values

You can display variable values by using the variable name within a `WriteLine()` method call. For example, Figure 2-1 shows a C# program that displays the value of the variable `someMoney`. Figure 2-2 shows the output of the program when executed at the command prompt.

```
using System;
class DisplaySomeMoney
{
    static void Main()
    {
        double someMoney = 39.45;
        Console.WriteLine(someMoney);
    }
}
```

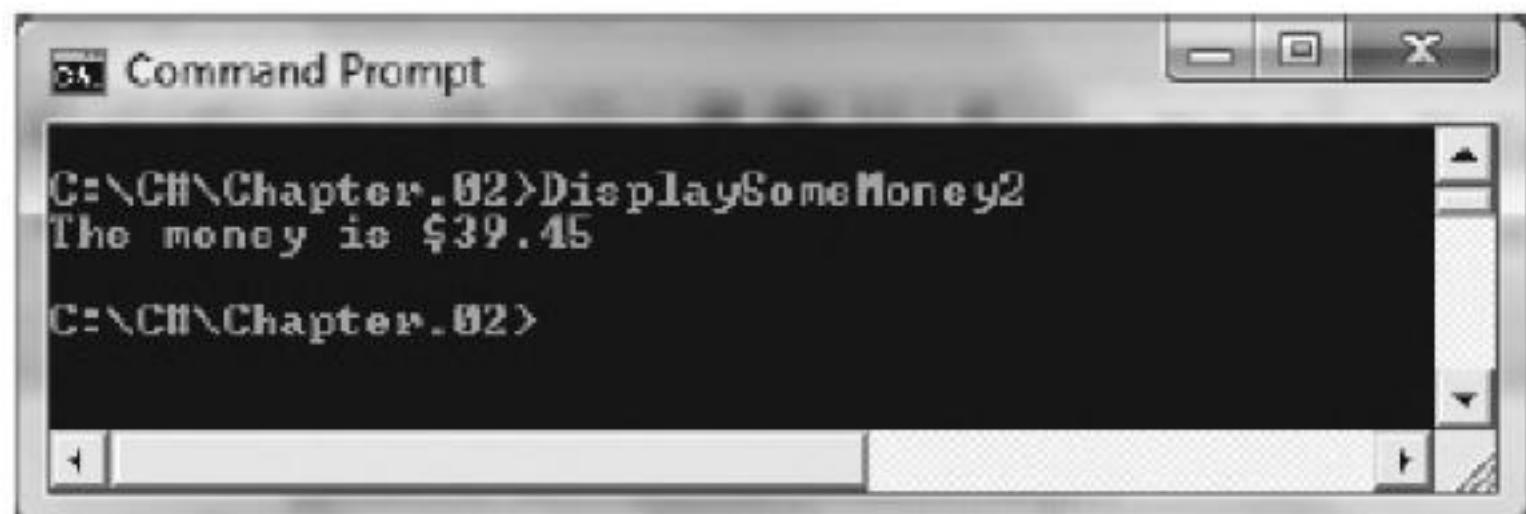
**Figure 2-1** Program that displays a variable value



**Figure 2-2** Output of `DisplaySomeMoney` program

```
using System;
class DisplaySomeMoney2
{
    static void Main()
    {
        double someMoney = 39.45;
        Console.Write("The money is $");
        Console.WriteLine(someMoney);
    }
}
```

**Figure 2-3** Program that displays a string and a variable value



**Figure 2-4** Output of DisplaySomeMoney2 program

If you want to display several strings and several variables, you can end up with quite a few `Write()` and `WriteLine()` statements. To make producing output easier, you can combine strings and variable values into a single `Write()` or `WriteLine()` statement in one of two ways: by concatenating them or by using a format string.

When you **concatenate** a string with another value, you join the values with a plus sign. For example, the following statement produces the same output as shown in Figure 2-4, but uses only one `WriteLine()` statement:

```
Console.WriteLine("The money is $" + someMoney);
```

A **format string** is a string of characters that optionally contains fixed text and contains one or more format items or placeholders for variable values. A **placeholder** consists of a pair of curly braces containing a number that indicates the desired variable's position in a list that follows the string. The first position is always position 0.



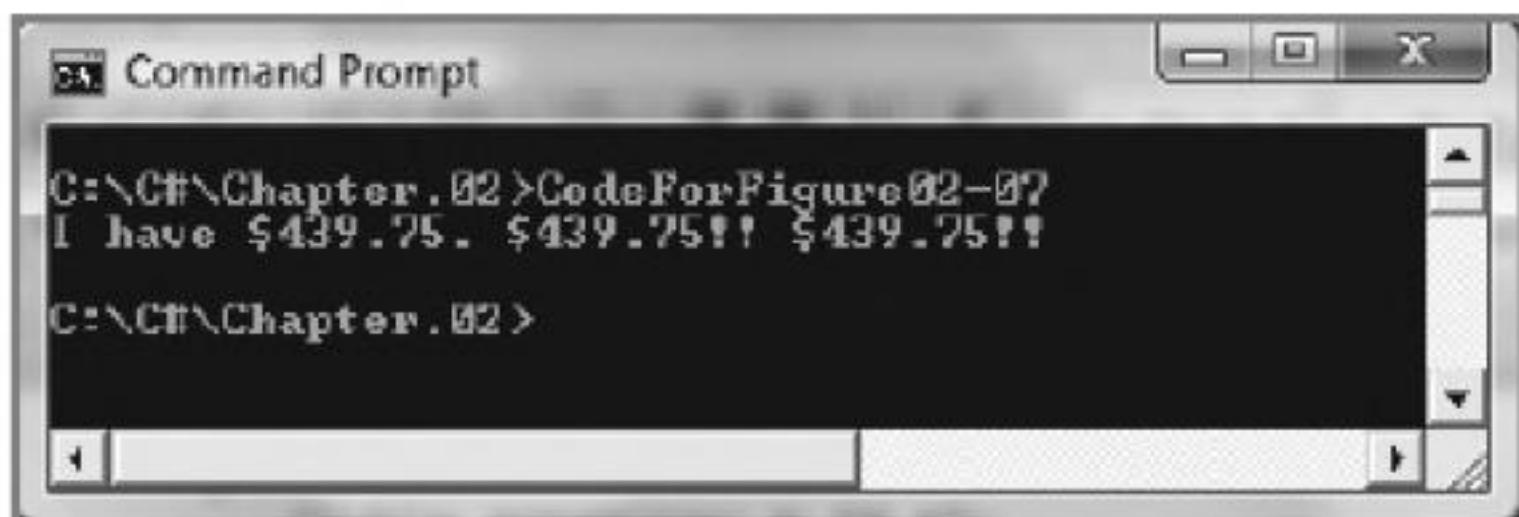
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

To display two variables within a single call to `Write()` or `WriteLine()`, you can use a statement like the following:

```
Console.WriteLine("The money is {0} and my age is {1}",  
    someMoney, myAge);
```

The number within the curly braces in the format string must be less than the number of values you list after the format string. In other words, if you list six values to be displayed, valid format position numbers are 0 through 5. You do not have to use the positions in order. For example, you can choose to display the value in position 2, then 1, then 0. You also can display a specific value multiple times. For example, if `someMoney` has been assigned the value 439.75, the following code produces the output shown in Figure 2-7:

```
Console.WriteLine("I have ${0}. ${0}!! ${0}!!",  
    someMoney);
```



**Figure 2-7** Displaying the same value multiple times

The output in Figure 2-7 could be displayed without using a format string, but you would have to name the `someMoney` variable three separate times as follows:

```
Console.WriteLine("I have $" + someMoney + ". $" + someMoney +  
    "!! $" + someMoney + "!!");
```

Besides the extra typing in this statement, you might find it harder to read than the version with the format string that includes a single set of quotation marks and a variable list at the end.

When you use a series of `WriteLine()` statements to display a list of variable values, the values are not automatically right-aligned as you normally expect numbers to be. For example, the following code produces the unaligned output shown in Figure 2-8:

```
Console.WriteLine("{0}", 4);  
Console.WriteLine("{0}", 56);  
Console.WriteLine("{0}", 789);
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

## TWO TRUTHS & A LIE

### Using the Integral Data Types

1. C# supports nine integral data types, including the most basic one, `int`.
2. Every Unicode character, including the letters of the alphabet and punctuation marks, can be represented as a number.
3. When you assign a value to any numeric variable, it is optional to use commas for values in the thousands.

The false statement is #3. When you assign a value to a numeric variable, you do not use commas, but you can type a plus or minus sign to indicate a positive or negative integer.



### You Do It

#### Declaring and Using Variables

In the following steps, you write a program that declares several integral variables, assigns values to them, and displays the results.

1. Open a new file. If you are using a simple text editor to write your C# programs, you will save the file as `DemoVariables.cs`. If you are using Visual Studio, select **Console Application**, name the project **DemoVariables**, and delete all the code in the program editing window before starting.
2. Create the beginning of a program that will demonstrate variable use. Use the `System` namespace, name the class **DemoVariables**, and type the class-opening curly brace.  
`using System;  
class DemoVariables  
{`

3. In the `Main()` method, declare two variables (an integer and an unsigned integer) and assign values to them.

```
static void Main()  
{  
    int anInt = -123;  
    uint anUnsignedInt = 567;
```

(continues)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

(continued)

Format Character	Description	Default Format (if no precision is given)
N or n	Number	[–]XX,XXX.XX
P or p	Percent	Represents a numeric value as a percentage
R or r	Round trip	Ensures that numbers converted to strings will have the same values when they are converted back into numbers
X or x	Hexadecimal	Minimum hexadecimal (base 16) representation

**Table 2-2** Format specifiers

You can use a format specifier with the `ToString()` method to convert a number into a string that has the desired format. For example, you can use the *F* format specifier to insert a decimal point to the right of a number that does not contain digits to the right of the decimal point, followed by the number of zeros indicated by the precision specifier. (If no precision specifier is supplied, two zeros are inserted.) For example, the first `WriteLine()` statement in the following code produces 123.00, and the second produces 123.000:

```
double someMoney = 123;
string moneyString;
moneyString = someMoney.ToString("F");
Console.WriteLine(moneyString);
moneyString = someMoney.ToString("F3");
Console.WriteLine(moneyString);
```



You will learn more about creating and using methods in the chapters called *Introduction to Methods* and *Advanced Method Concepts*.

You use *C* as the format specifier when you want to represent a number as a currency value. Currency values appear with a dollar sign, appropriate commas, and the desired number of decimal places; negative values appear within parentheses. The integer you use following the *C* indicates the number of decimal places. If you do not provide a value for the number of decimal places, then two digits are shown after the decimal separator by default. For example, both of the following `WriteLine()` statements produce \$456,789.00:

```
double moneyValue = 456789;
string conversion;
conversion = moneyValue.ToString("C");
Console.WriteLine(conversion);
conversion = moneyValue.ToString("C2");
Console.WriteLine(conversion);
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The `+=` operator is the **add and assign operator**; it adds the operand on the right to the operand on the left and assigns the result to the operand on the left in one step. For example, if `payRate` is 20, then `payRate += 5` results in `payRate` holding the value 25. Similarly, the following statements both increase `bankBal` by a rate stored in `interestRate`:

```
bankBal += bankBal * interestRate;  
bankBal = bankBal + bankBal * interestRate;
```

For example, if `bankBal` is 100.00 and `interestRate` is 0.02, then both of the previous statements result in `bankBal` holding 102.00.

Additional shortcut operators in C# are `-=`, `*=`, and `/=`. Each of these operators is used to perform an operation and assign the result in one step. You cannot place spaces between the two characters that compose these operators; spaces preceding or following the operators are optional. Examples in which the shortcut operators are useful include the following:

- `balanceDue -= payment` subtracts a payment from `balanceDue` and assigns the result to `balanceDue`.
- `rate *= 100` multiplies `rate` by 100 and assigns the result to `rate`. For example, it converts a fractional value stored in `rate`, such as 0.27, to a whole number, such as 27.
- `payment /= 12` changes a payment value from an annual amount to a monthly amount due.

When you want to increase a variable's value by exactly 1, you can use either of two other shortcut operators—the **prefix increment operator** and the **postfix increment operator**.

To use a prefix increment operator, you type two plus signs before the variable name.

For example, these statements result in `someValue` holding 7:

```
int someValue = 6;  
++someValue;
```

The variable `someValue` holds 1 more than it held before the `++` operator was applied. To use a postfix `++`, you type two plus signs just after a variable name. Executing the following statements results in `anotherValue` holding 57:

```
int anotherValue = 56;  
anotherValue++;
```

You can use the prefix `++` and postfix `++` with variables, but not with constants. An expression such as `++84` is illegal because 84 is constant and must always remain as 84. However, you can create a variable as in `int val = 84;`, and then write `++val` or `val++` to increase the variable's value to 85.

The prefix and postfix increment operators are **unary operators** because you use them with one operand. Most arithmetic operators, like those used for addition and multiplication, are binary operators that operate on two operands.

When you only want to increase a variable's value by 1, there is no apparent difference between using the prefix and postfix increment operators. However, these operators function differently. When you use the prefix `++`, the result is calculated and stored, and then the



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
string word = "water";
Position: 0 1 2 3 4


|   |   |   |   |   |
|---|---|---|---|---|
| w | a | t | e | r |
|---|---|---|---|---|


Start position      Length
word.Substring(0, 1) is "w"
word.Substring(0, 2) is "wa"
word.Substring(1, 2) is "at"
word.Substring(2, 3) is "ter"
word.Substring(0, word.Length) is "water"
word.Substring(3, 4) produces an error message
```

**Figure 2-19** Using the Substring() method

The **StartsWith()** method is another useful string method. In the example that defines `word` as “water”, the expression `word.StartsWith("wa")` is true, and the expression `word.StartsWith("waet")` is false.

## TWO TRUTHS & A LIE

### Using the `string` Data Type

1. If `creature1 = "dog"` and `creature2 = "cat"`, then the value of `String.Equals(creature1, creature2)` is `false`.
2. If `creature1 = "dog"` and `creature2 = "cat"`, then the value of `String.Compare(creature1, creature2)` is `false`.
3. If `creature1 = "dog"` and `creature2 = "cat"`, then the value of `creature1.CompareTo(creature2)` is a positive number.

The `false` statement is #2. If `creature1 = "dog"` and `creature2 = "cat"`, then the value of `String.Compare(creature1, creature2)` is a positive number.

## Defining Named Constants

By definition, a variable’s value can vary, or change. Sometimes you want to create a **named constant**, an identifier for a memory location whose contents cannot change. You create a named constant by adding the keyword `const` before the data type in a declaration. Although there is no requirement to do so, programmers usually name constants using all uppercase letters, inserting underscores for readability. This convention makes constant names stand



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The `Console.Read()` method is similar to the `Console.ReadLine()` method. `Console.Read()` reads just one character from the input stream, whereas `Console.ReadLine()` reads every character in the input stream until the user presses the Enter key.

If you want to use the input value as a type other than `string`, then you must use a conversion method to change the input `string` to the proper type. Two methods are available for converting strings—the `Convert` class and `Parse()` methods.

## Using the Convert Class

Figure 2-20 shows an interactive program that prompts the user for a price and calculates a 6 percent sales tax. The program displays “Enter the price of an item” on the screen. Such an instruction to the user is a **prompt**. Notice that the program uses a `Write()` statement for the prompt instead of a `WriteLine()` statement so that the user’s input appears on the screen on the same line as the prompt. Also notice that the prompt contains a space just before the closing quote so that the prompt does not crowd the user’s input on the screen. After the prompt appears, the `Console.ReadLine()` statement accepts a string of characters and assigns them to the variable `itemPriceAsString`. Before the tax can be calculated, this value must be converted to a number. This conversion is accomplished using the `Convert` class `ToDouble()` method in the shaded statement. Figure 2-21 shows a typical execution of the program in which the user typed 28.77 as the input value.

```
using System;
class InteractiveSalesTax
{
    static void Main()
    {
        const double TAX_RATE = 0.06;
        string itemPriceAsString;
        double itemPrice;
        double total;
        Console.Write("Enter the price of an item >> ");
        itemPriceAsString = Console.ReadLine();
        itemPrice = Convert.ToDouble(itemPriceAsString);
        total = itemPrice * TAX_RATE;
        Console.WriteLine("With a tax rate of {0}, a {1} item " +
            "costs {2} more.", TAX_RATE, itemPrice.ToString("C"),
            total.ToString("C"));
    }
}
```

**Figure 2-20** InteractiveSalesTax program



In the program in Figure 2-20, the angle brackets at the end of the prompt are not required; they are merely cosmetic. You might prefer other punctuation such as a colon, ellipsis, or hyphen to indicate that input is required.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

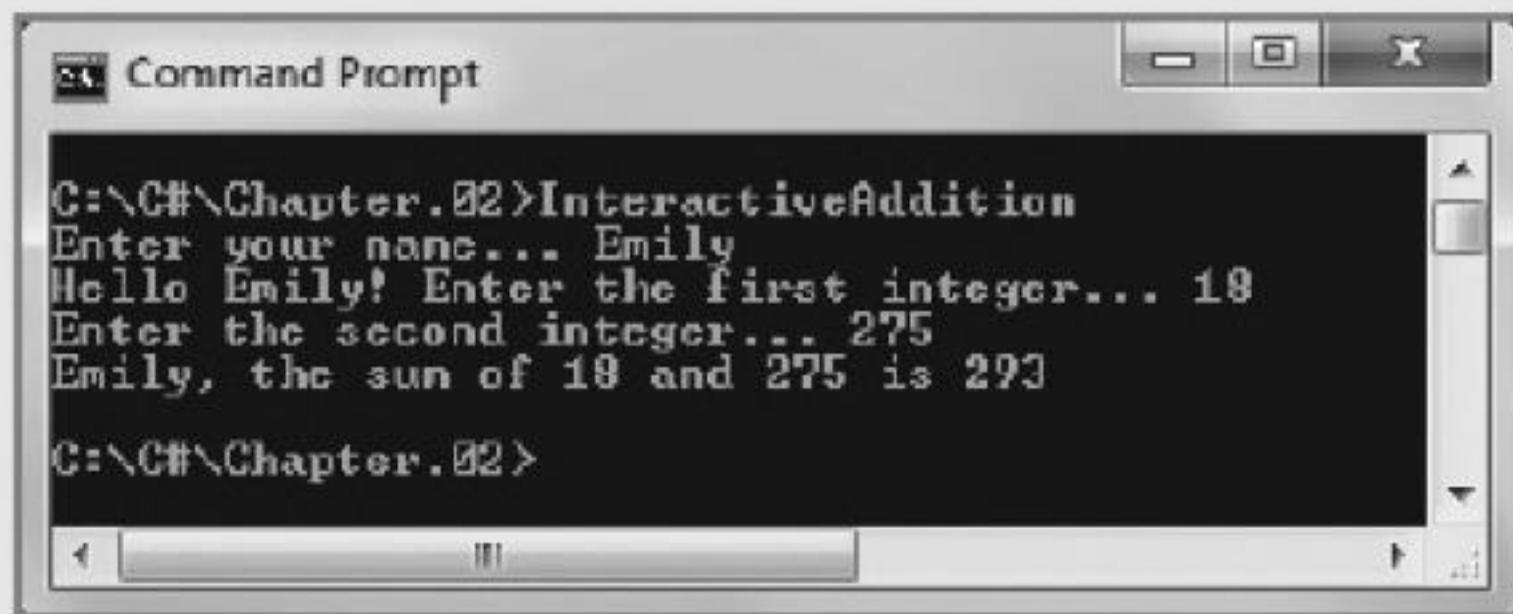


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

(continued)

```
    sum = first + second;
    Console.WriteLine("{0}, the sum of {1} and {2} is {3}",
                      name, first, second, sum);
}
}
```

7. Save, compile, and run the program. When prompted, supply your name and any integers you want, and confirm that the result appears correctly. Figure 2-22 shows a typical run of the program.



**Figure 2-22** Typical execution of InteractiveAddition program

## Chapter Summary

- A constant is a data value that cannot be changed after a program is compiled; a value in a variable can change. C# provides for 15 basic built-in types of data. A variable declaration includes a data type, an identifier, an optional assigned value, and a semicolon.
- You can display variable values by using the variable name within a `WriteLine()` or `Write()` method call. To make producing output easier, you can combine strings and variable values into a single `Write()` or `WriteLine()` statement by using a format string.
- C# supports nine integral data types: `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, and `char`.
- C# supports three floating-point data types: `float`, `double`, and `decimal`. You can use format and precision specifiers to display floating-point data to a specified number of decimal places.
- You use the binary arithmetic operators `+`, `-`, `*`, `/`, and `%` to manipulate values in your programs. Multiplication, division, and remainder always take place prior to addition or subtraction in an expression, unless you use parentheses to override the normal precedence. C# provides several shortcut arithmetic operators, including the binary operators `+=`, `-=`, `*=`, `/=`, and the unary prefix and postfix increment (`++`) and decrement (`--`) operators.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

2. What is the value of each of the following Boolean expressions?
  - a.  $6 < 8$
  - b.  $7 \leq 7$
  - c.  $6 * 2 > 10$
  - d.  $4 == 5$
  - e.  $1 + 8 \leq 9$
  - f.  $3 + 5 == 33 / 4$
  - g.  $7 != 6$
  - h.  $7 != 7$
  - i.  $3 == 4 - 1$
  - j.  $3 * 4 / 5 == 2$
3. Choose the best data type for each of the following, so that no memory storage is wasted. Give an example of a typical value that would be held by the variable and explain why you chose the type you did.
  - a. the number of siblings you have
  - b. your final grade in this class
  - c. the population of Earth
  - d. the number of passengers on a bus
  - e. one player's score in a Scrabble game
  - f. the year an historical event occurred
  - g. the number of legs on an animal
  - h. one team's score in a Major League Baseball game
4. In this chapter, you learned that although a `double` and a `decimal` both hold floating-point numbers, a `double` can hold a larger value. Write a C# program named **DoubleDecimalTest** that declares and displays two variables—a `double` and a `decimal`. Experiment by assigning the same constant value to each variable so that the assignment to the `double` is legal but the assignment to the `decimal` is not. In other words, when you leave the `decimal` assignment statement in the program, an error message should be generated that indicates the value is outside the range of the type `decimal`, but when you comment out the `decimal` assignment and its output statement, the program should compile correctly.
5. a. Write a C# program named **MilesToFeet** that declares a named constant that holds the number of feet in a mile: 5280. Also declare a variable to represent the distance in miles between your house and your uncle's house. Assign an appropriate value to the variable—for example, 8.5. Compute and display the distance to your uncle's house in both miles and feet. Display explanatory text with the values—for example, *The distance to my uncle's house is 8.5 miles or 44880 feet.*  
b. Convert the **MilesToFeet** class to an interactive application named **MilesToFeetInteractive**. Instead of assigning a value to the distance, accept the value from the user as input.
6. a. Write a C# program named **ProjectedSales** that includes a named constant representing next year's anticipated 10 percent increase in sales for each division of a company. (You can represent 10 percent as 0.10). Also declare



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

# 3

## CHAPTER

# Using GUI Objects and the Visual Studio IDE

(Screen shots are used with permission from Microsoft.)

In this chapter you will:

- ◎ Create a Form in the Visual Studio IDE
- ◎ Use the Toolbox to add a Button to a Form
- ◎ Add Labels and TextBoxes to a Form
- ◎ Name Forms and controls
- ◎ Correct errors
- ◎ Decide which interface to use



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



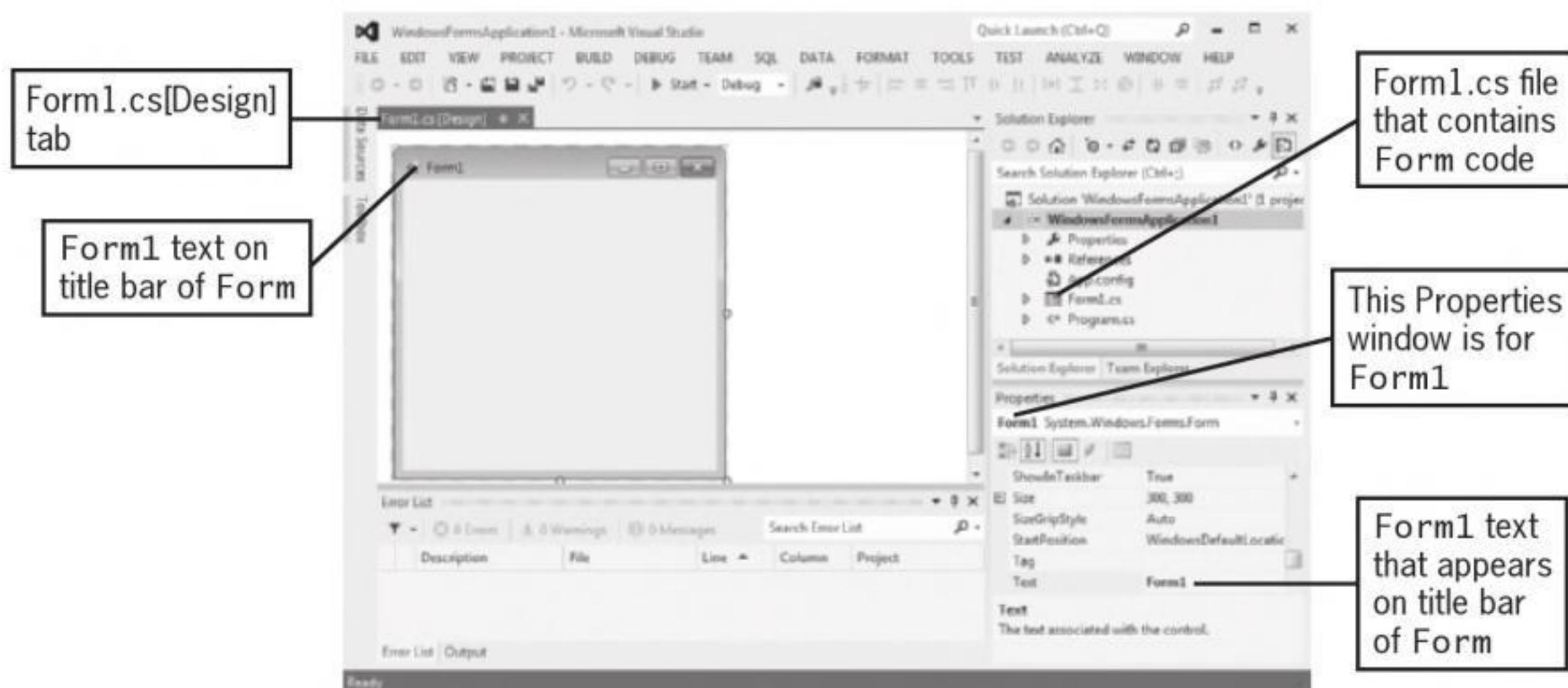
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

When you create a Windows Forms project, Visual C# adds a form to the project and calls it **Form1**. You can see **Form1** in the following locations in Figure 3-3:

- On the folder tab at the top of the Form Designer area (followed by *[Design]*)
- In the title bar of the form in the Form Designer area
- In the Solution Explorer file list (with a .cs extension)



**Figure 3-3** Displaying the properties of Form1

When you click the **Form** in the Form Designer area, you see its name in two more locations: in the Properties window in the lower-right corner, and as the **Text** property of the form that is in the Form Designer area. You can scroll through a list of properties in the Properties window. If you click a property, a brief description of it appears at the bottom of the Properties window. For example, Figure 3-3 displays a description of the **Text** property of **Form1**.

You can change the appearance, size, color, and window management features of a **Form** by setting its properties. The **Form** class contains approximately 100 properties; Table 3-1 lists a few of the **Form** features you can change. For example, setting the **Text** property allows you to specify the caption of the **Form** in the title bar, and setting the **Size**, **BackColor**, and **ForeColor** allow you to further customize the **Form**. The two left buttons at the top of the Properties list allow you to organize properties by category or alphabetically. The properties shown in Figure 3-3 are in alphabetical order. Not every property that can be used with a **Form** appears in the Properties window in the Visual Studio IDE—only the most frequently used properties are listed.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



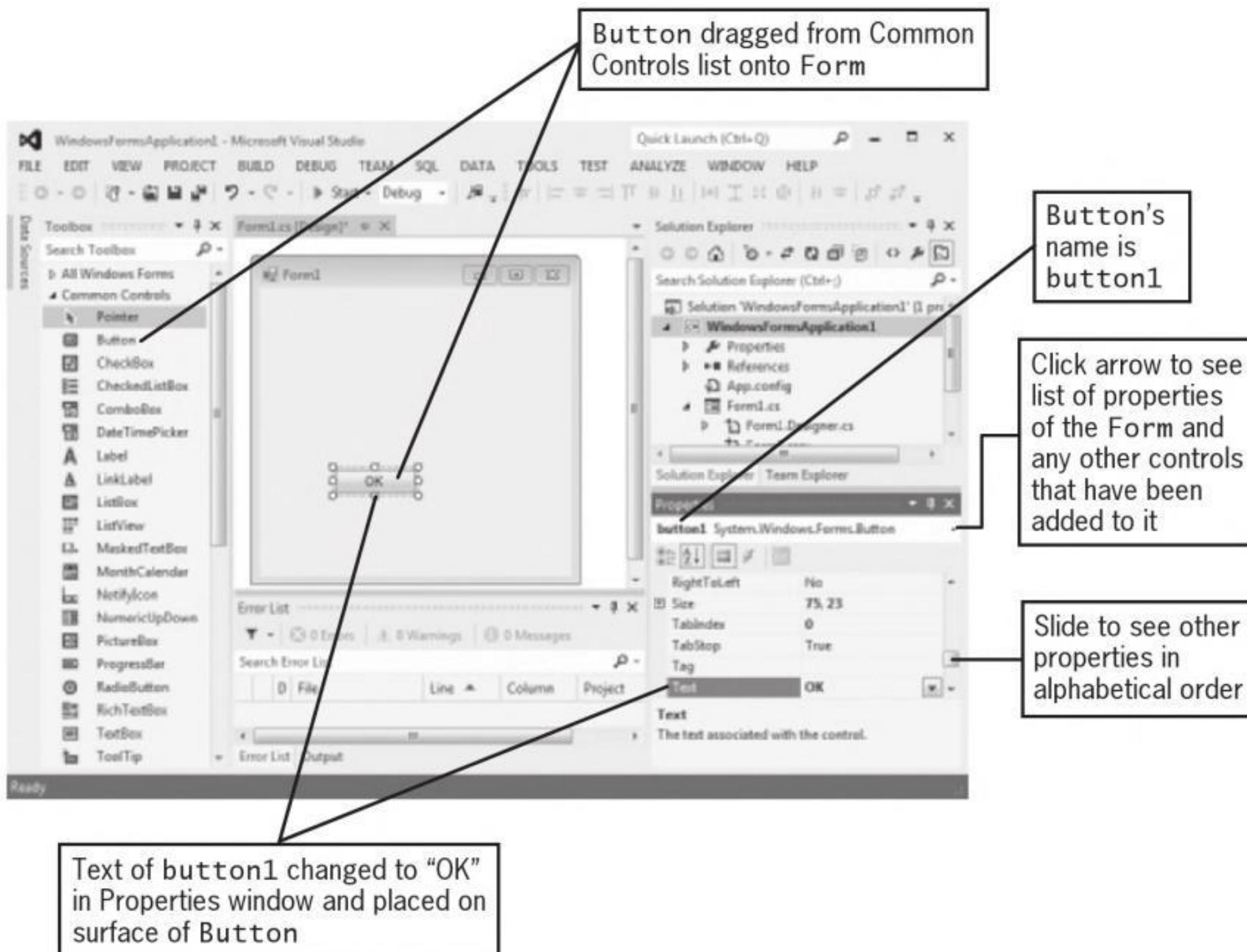
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

## Using the Toolbox to Add a Button to a Form

111



**Figure 3-5** A Button dragged onto a Form

In Figure 3-5, the programmer has clicked `button1`, so the Properties window shows the properties for `button1`. (If you click the Form, the Properties window changes to show Form properties instead of Button properties.) The `Text` property of `button1` has been changed to “OK”, and the button on the Form has handles that can be used to resize it. You can scroll through the Properties list and examine other properties that can be modified for the Button. For example, you can change its `BackColor`, `ForeColor`, `Font`, and `Size`. Table 3-2 describes a few of the properties available for Button objects. This chapter discusses only a few properties of each control. You can learn about other properties in the chapter *Using Controls*.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You can write any statements you want between the curly braces of the `button1_Click()` method. For example, you can declare variables, perform arithmetic statements, and produce output. You also can include block or line comments. The next sections of this chapter include several statements added to a `Click()` method.



Watch the video *Creating a Functional Button*.

### TWO TRUTHS & A LIE

#### Using the Toolbox to Add a Button to a Form

1. When a user clicks a **Button**, the action fires a click event that causes the **Button's Click()** method to execute.
2. If a **Button's identifier** is `reportButton`, then the name of its `Click()` method is `reportButton.Click()`.
3. You can write any statements you want between the curly braces of a **Button's Click()** method.

The false statement is #2. If a **Button's identifier** is `reportButton`, then the name of its `Click()` method is `reportButton_Click()`.

## Adding Labels and TextBoxes to a Form

Suppose that you want to create an interactive GUI program that accepts two numbers from a user and outputs their sum when the user clicks a **Button**. To provide prompts for the user and to display output, you can add **Labels** to a **Form**, and to get input from a user, you can provide **TextBoxes**.

**Labels** are controls that you use to display text to communicate with an application's user. Just like a **Button**, you can drag a **Label** onto a **Form**, as shown in Figure 3-7. By default, the first **Label** you drag onto a **Form** is named `label1`, and the text that appears on the **Label** also is `label1`. You can change its **Text** property to display any string of text; depending on the amount of text you enter, you might need to change the size of the **Label** by dragging its resizing handles or altering its **Size** property. In Figure 3-7, “Enter a number” has been assigned to `label1`'s **Text** property. If you want to create multiple



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



**Figure 3-11** The Form when it first appears and after a user has entered two integers and clicked the Button

You can change the values in either or both of the TextBoxes in the Form and click the Button to see a new result. When you finish using the application, you can close it by clicking the Close button in the upper-right corner of the Form.

## Understanding Focus and Tab Control

When a GUI program displays a Form, one of the components on the Form has focus. The control with **focus** is the one that can receive keyboard input. When a Button has focus, a thin, bright line appears around it to draw your attention, and the Button's associated event executes when you click the button or press the Enter key on the keyboard. When a TextBox has focus, a blinking cursor appears inside it.

When an application is running, the user can give focus to a component by clicking it, or can switch focus from one component to another using the Tab key. The order in which controls receive focus from successive Tab key presses is their **tab order**. By default, controls are assigned tab order based on the order in which you place them on a Form when designing a program. The first control you place on a Form has TabIndex 0, the next control has TabIndex 1, and so on. When you start a program, the control with TabIndex 0 automatically has focus. You can check a control's tab order by viewing its TabIndex property in the Properties list of the IDE. If you do not want a control to be part of the tabbing sequence for a Form, you can change its TabStop property from true to false in the Properties list.

You can change a control's TabIndex value by changing the entry next to TabIndex in the Properties list of the IDE. Alternatively, you can select VIEW from the main menu, and then select Tab Order. You see a representation of the Form with a small blue number next to each



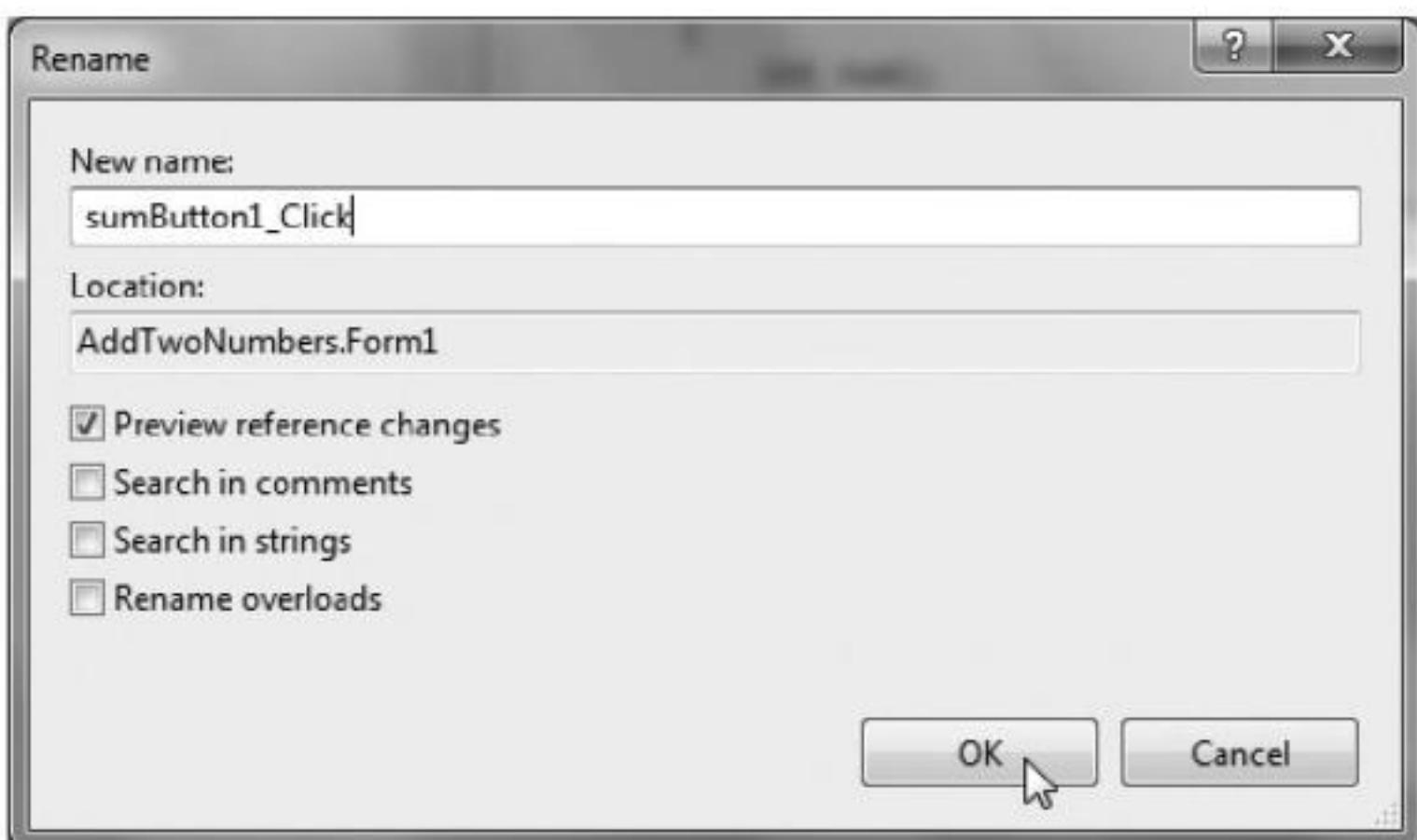
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



**Figure 3-13** The Rename dialog box

## TWO TRUTHS & A LIE

### Naming Forms and Controls

1. Label and TextBox names usually start with an uppercase letter, but Form names usually start with a lowercase letter.
2. Professional programmers usually do not retain the default names for Forms and controls.
3. If you rename a control after you have created an event for it, you must refactor the code.

The false statement is #1. Label and TextBox names usually start with a lowercase letter, but Form names usually start with an uppercase letter.

## Correcting Errors

Just like in console-based programs, you will often generate syntax errors when you use the visual developer to create GUI applications. If you build or run a program that contains a syntax error, you see *Build failed* in the lower-left corner of the IDE. You also see an error dialog box like the one shown in Figure 3-14. When you are developing a program, you should always click No in response to “Would you like to continue and run the last successful build?” Clicking Yes will run the previous version of the program you created before inserting the mistake that caused the dialog box to appear. Instead, click No, examine the error messages and code, and correct the mistake.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

## Deciding Which Interface to Use

You have learned to create console applications in which most of the action occurs in a `Main()` method and the `WriteLine()` and `ReadLine()` methods are used for input and output. You also have learned to create GUI applications in which most of the action occurs within an event-handling method such as a `Click()` method, and `Labels` and `TextBoxes` are used for input and output. Both types of applications can use declared variables and constants, and, as you will learn in the next few chapters, both types of programs can contain the basic building blocks of applications—decisions, loops, arrays, and calls to other methods. When you want to write a program that displays a greeting or sums some integers, you can do so using either type of interface and employing the same logic to get the same results. So, should you develop programs using a GUI interface or a console interface?

- GUI applications look “snazzier.” It is easy to add colors and fonts to them, and they contain controls that a user can manipulate with a mouse. Also, users are accustomed to GUI applications from their experiences on the Web. However, GUI programs usually take longer to develop than their console counterparts because you can spend a lot of time setting up the controls, placing them on a `Form`, and adjusting their sizes and relative positions before you write the code that does the actual work of the program. GUI applications created in the IDE also require much more disk space to store.



Designing aesthetically pleasing, functional, and user-friendly `Forms` is an art; entire books are devoted to the topic.

- Console applications look dull in comparison to GUIs, but they can often be developed more quickly because you do not spend much time setting up objects for a user to manipulate. When you are learning programming concepts like decision-making and looping, you might prefer to keep the interface simple so that you can better concentrate on the new logical constructs being presented.

In short, it doesn’t matter which interface you use to develop programs while learning the intricacies of the C# programming language. In the following “You Do It” section, you develop an application that is similar to the one you developed at the console in Chapter 1. In the programming exercises at the end of this chapter, you will develop programs that are identical in purpose to programs written using the console in the chapter called *Using Data*. Throughout the next several chapters on decisions, looping, and arrays, many concepts will be illustrated in console applications, because the programs are shorter and the development is simpler. However, you also will occasionally see the same concept illustrated in a program that uses a GUI interface, to remind you that the program logic is the same no matter which interface is used. After you complete the chapters *Using Classes and Objects*, *Using Controls*, and *Handling Events*, you will be able to write even more sophisticated GUI applications.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

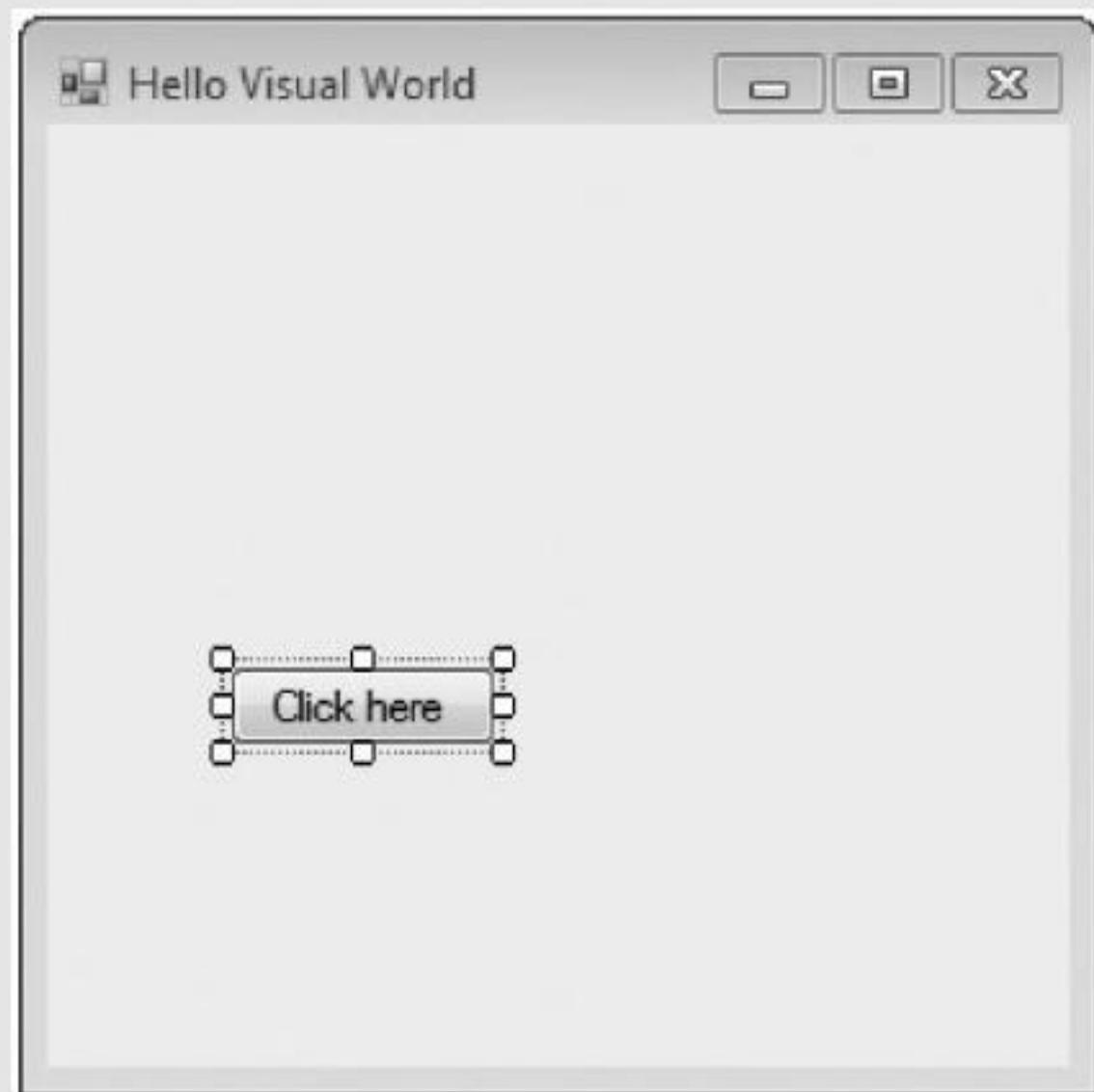


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

(continued)

5. Examine the Toolbox on the left side of the screen. Select **Common Controls** if it is not already selected. In the Toolbox, click **Button**. As you move your mouse off the Toolbox and onto the form, the mouse pointer changes so that it appears to carry a Button. Position your mouse anywhere on the form, then release the mouse button. The Button appears on the Form and contains the text "button1". When you click the Button, handles appear that you can drag to resize it. When you click off the Button on the Form, the handles disappear. Click the Button to display the properties for button1. Change its Name to **displayOutputButton**, and change its Text property to **Click here**. When you press Enter on the keyboard or click anywhere on the Form, the text of the Button on the Form changes to "Click here". See Figure 3-19.

131



**Figure 3-19** A Form with changed Text and a Button

6. Scroll through the other **displayOutputButton** properties. For example, examine the **Location** property. The first value listed for **Location** indicates horizontal position, and the second value indicates vertical position. Drag the Button across the Form to a new position. Each time

(continues)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

## Chapter Summary

- Forms are GUI objects that provide an interface for collecting, displaying, and delivering information. Forms almost always include controls such as labels, text boxes, and buttons that users can manipulate to interact with a program. Every Form and control on a Form has multiple properties you can set using the IDE.
- The Toolbox displays a list of available controls you can add to a Form. The list includes Button, CheckBox, and Label. From the Toolbox, you can drag controls onto a Form where they will be most useful. After you have dragged a Button onto a Form, you can double-click it to create the method that executes when a user clicks the Button, and you can write any statements you want between the curly braces of the Click() method.
- Labels are controls that you use to display text to communicate with an application's user. TextBoxes are controls through which a user can enter input data in a GUI application. Both have a Text property; frequently an application starts with the Text empty for TextBoxes. Because a user might type any value, the value in a TextBox is a string by default.
- Usually, you want to provide reasonable Name property values for all the controls you place on a Form. Although any identifier that starts with a letter is legal, by convention you should start control names with a lowercase letter and use camel casing as appropriate, start Form names with an uppercase letter and use camel casing as appropriate, and use the type of object in the name.
- If you build or run a program that contains a syntax error, you see *Build failed* in the lower-left corner of the IDE and an error dialog box. Errors are shown in the error list at the bottom of the screen. The error list also shows the file, line, position in the line, and project in which the error occurred. If you double-click the error message, the cursor is placed at the location in the code where the error was found. If you inadvertently create an event-handling method that you do not want, you should eliminate the event using the Properties window in the IDE. The ultimate authority on C# classes is the Visual Studio Help documentation.
- Both console and GUI applications can contain variables and constants, decisions, loops, arrays, and calls to other methods. GUI applications look “snazzier,” and they contain controls that a user can manipulate with a mouse. However, GUI programs usually take longer to develop than their console counterparts. When writing your own programs, you will use the interface you prefer or one that your instructor or boss requires.

135

## Key Terms

The **interface** is the environment a user sees when a program executes.

**Forms** are GUI objects that provide an interface for collecting, displaying, and delivering information.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

## Exercises



## Programming Exercises

1. Write a GUI program named **DistanceGUI** that allows the user to input a distance in miles and output the value in feet. There are 5280 feet in a mile.



The exercises in this section should look familiar to you. Each is similar to an exercise in Chapter 2, where you created solutions using console input and output.

2. Write a GUI program named **ProjectedSalesGUI** that allows a user to enter the current year's sales figures for the North and South divisions of a company. Compute and display, with explanatory text, next year's projected sales for each division, which reflect 10 percent increases.
  3. Write a GUI program named **EggsInteractiveGUI** that allows a user to input the number of eggs produced in a month by each of five chickens. Sum the eggs, then display the total in dozens and eggs. For example, a total of 127 eggs is 10 dozen and 7 eggs. Figure 3-21 shows a typical execution.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

# 4

## CHAPTER

# Making Decisions

(Screen shots are used with permission from Microsoft.)

In this chapter you will:

- ◎ Understand logic-planning tools and decision making
- ◎ Learn how to make decisions using the `if` statement
- ◎ Learn how to make decisions using the `if-else` statement
- ◎ Use compound expressions in `if` statements
- ◎ Make decisions using the `switch` statement
- ◎ Use the conditional operator
- ◎ Use the NOT operator
- ◎ Learn to avoid common errors when making decisions
- ◎ Learn about decision-making issues in GUI programs



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Boolean values; every computer decision results in a Boolean value. For example, program code that you write never includes a question like “What number did the user enter?” Instead, the decisions might be: “Did the user enter a 1?” “If not, did the user enter a 2?” “If not, did the user enter a 3?”

147

## TWO TRUTHS & A LIE

### Understanding Logic-Planning Tools and Decision Making

1. A sequence structure has three or more alternative logical paths.
2. A decision structure involves choosing between alternative courses of action based on some value within a program.
3. When reduced to their most basic form, all computer decisions are yes-or-no decisions.

The **false** statement is #1. In a sequence structure, one step follows another unconditionally.

## Making Decisions Using the **if** Statement

The **if** and **if-else** statements are the two most commonly used decision-making statements in C#. You use an **if statement** to make a single-alternative decision. In other words, you use an **if** statement to determine whether an action will occur. The **if** statement takes the following form:

```
if(testedExpression)
    statement;
```

where *testedExpression* represents any C# expression that can be evaluated as **true** or **false** and *statement* represents the action that will take place if the expression evaluates as **true**. You must place the **if** statement’s evaluated expression between parentheses. You can leave a space between the keyword **if** and the opening parenthesis if you think that format is easier to read.

Usable expressions in an **if** statement include Boolean expressions such as `amount > 5` and `month == "May"` as well as the value of `bool` variables such as `isValidIDNumber`. If the expression evaluates as **true**, then the statement executes. Whether the expression evaluates as **true** or **false**, the program continues with the next statement following the complete **if** statement.



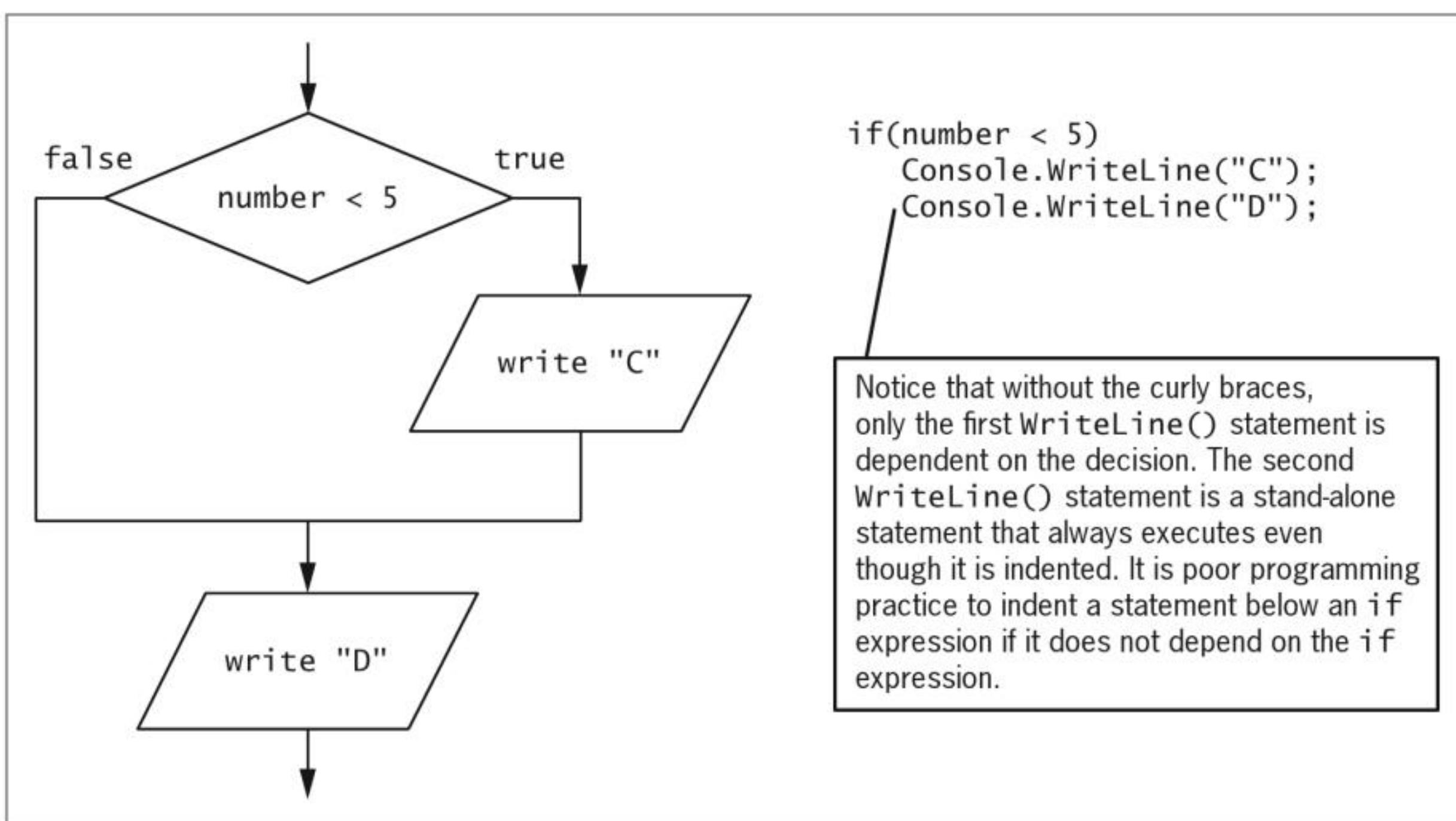
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



**Figure 4-6** Flowchart and code including an **if** statement that is missing curly braces or that has inappropriate indenting

When you create a block using curly braces, you do not have to place multiple statements within it. It is perfectly legal to block a single statement. Blocking a single statement can be a useful technique to help prevent future errors. When you write **if** statements that use curly braces even when they aren't required—and then a program later is modified to include multiple statements that depend on the **if**—you will naturally place the additional statements within the block that is already in place. Creating a block with braces that contain no statements also is legal in C#. You usually do so only when starting to write a program, as a reminder to yourself to add statements later.



In C#, it is customary to align a block's opening and closing braces. Some programmers prefer to place the opening brace on the same line as the **if** expression instead of giving the brace its own line. This style is called the **K & R style**, named for Brian Kernighan and Dennis Ritchie, who wrote the first book on the C programming language.

You can place any number of statements within a block contained by curly braces, including other **if** statements. Of course, if a second **if** statement is the only statement that depends on the first **if**, then no braces are required. Figure 4-7 shows the logic for a **nested if** statement in which one decision structure is contained within another. With a nested **if** statement, a second **if**'s Boolean expression is tested only when the first **if**'s Boolean expression evaluates as **true**.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

## Making Decisions Using the **if-else** Statement

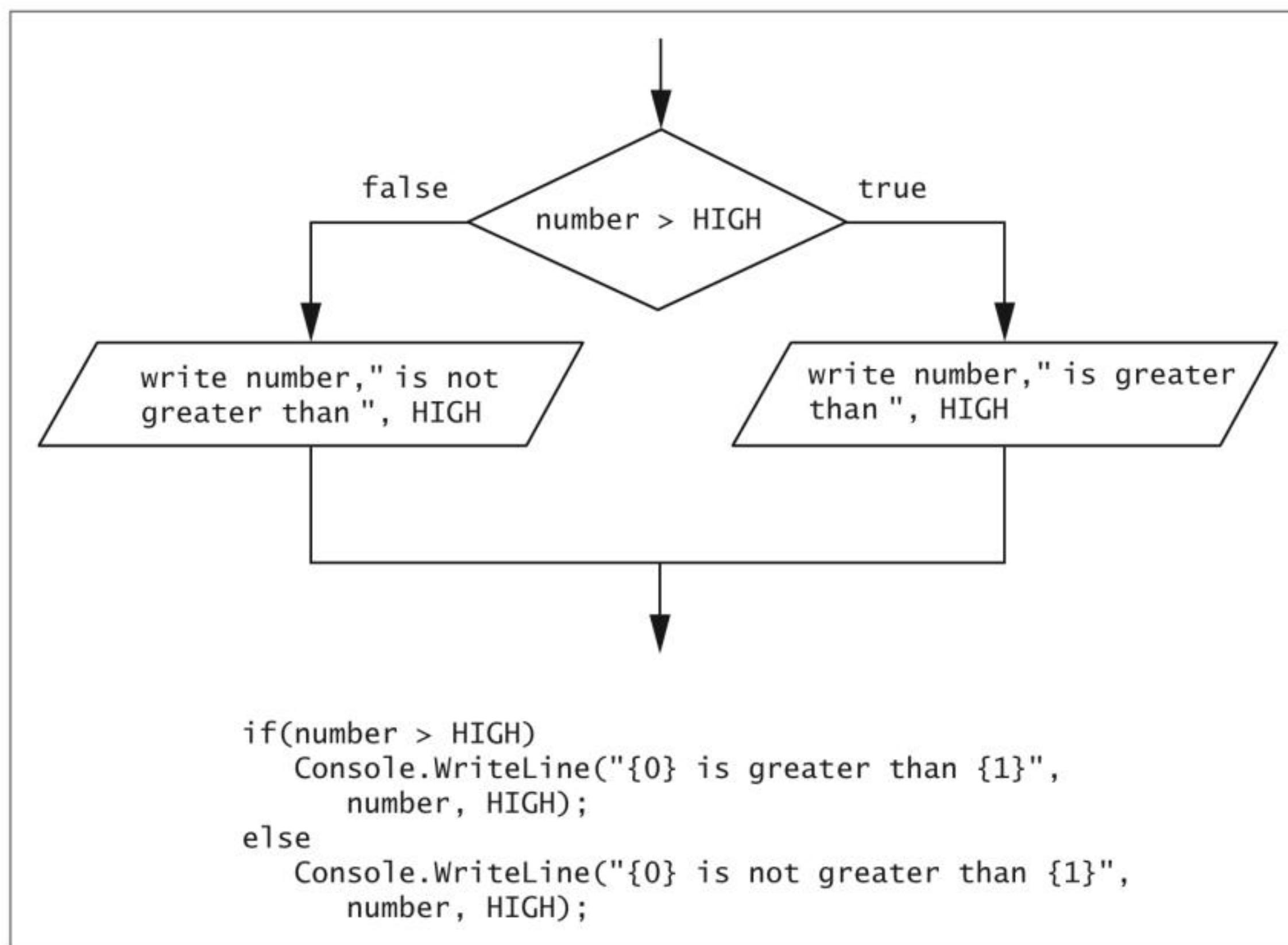
Some decisions you make are **dual-alternative decisions**; they have two possible resulting actions. If you want to perform one action when a Boolean expression evaluates as **true** and an alternate action when it evaluates as **false**, you can use an **if-else statement**. The **if-else** statement takes the following form:

```
if(expression)
    statement1;
else
    statement2;
```

You can code an **if** without an **else**, but it is illegal to code an **else** without an **if**.

Just as you can block several statements so they all execute when an expression within an **if** is **true**, you can block multiple statements after an **else** so that they will all execute when the evaluated expression is **false**.

For example, Figure 4-10 shows the logic for an **if-else** statement, and Figure 4-11 shows a program that contains the statement. With every execution of the program, one or the other of the two **WriteLine()** statements executes. The indentation shown in the **if-else** example in Figure 4-11 is not required, but it is standard. You vertically align the keyword **if** with the keyword **else**, and then indent the action statements that depend on the evaluation. Figure 4-12 shows two executions of the program.



**Figure 4-10** Flowchart and code showing the logic of a dual-alternative **if-else** statement



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

(continued)

```
    else
        if(num2 == num3)
            Console.WriteLine("Last two are equal");
```

7. Finally, if none of the pairs (num1 and num2, num1 and num3, or num2 and num3) is equal, display an appropriate message. For clarity, the `else` should vertically align under `if(num2 == num3)`.

```
    else
        Console.WriteLine
            ("No two numbers are equal");
```

8. Add a closing curly brace for the `Main()` method and a closing curly brace for the class.
9. Save and compile the program, and then execute it several times, providing different combinations of equal and nonequal integers when prompted. Figure 4-13 shows several executions of the program.

```
C:\> Command Prompt
C:\> C:\>Chapter.04>CompareThreeNumbers
Enter an integer 4
Enter an integer 16
Enter an integer 8
No two numbers are equal

C:\> C:\>Chapter.04>CompareThreeNumbers
Enter an integer 12
Enter an integer 45
Enter an integer 12
First and last are equal

C:\> C:\>Chapter.04>CompareThreeNumbers
Enter an integer 67
Enter an integer 5
Enter an integer 5
Last two are equal

C:\> C:\>Chapter.04>
```

**Figure 4-13** Several executions of the `CompareThreeNumbers` program



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



Watch the video *Using the AND and OR Operators*.

## Using the Logical AND and OR Operators

C# provides two logical operators that you generally do *not* want to use when making comparisons. However, you should learn about them because they might be used in programs written by others, and because you might use one by mistake when you intend to use a conditional operator.

The **Boolean logical AND operator** ( `&` ) and **Boolean logical inclusive OR operator** ( `|` ) work just like their `&&` and `||` (*conditional* AND and OR) counterparts, except they do not support short-circuit evaluation. That is, they always evaluate both sides of the expression, no matter what the first evaluation is. This can lead to a **side effect**, or unintended consequence. For example, in the following statement that uses `&&`, if `salesAmountForYear` is not at least 10,000, the first half of the expression is `false`, so the second half of the Boolean expression is never evaluated and `yearsOfService` is not increased.

```
if(salesAmountForYear >= 10000 && ++yearsOfService > 10)
    bonus = 200;
```

On the other hand, when a single `&` is used and `salesAmountForYear` is not at least 10,000, then even though the first half of the expression is `false`, the second half is still evaluated, and `yearsOfService` is increased:

```
if(salesAmountForYear >= 10000 & ++yearsOfService > 10)
    bonus = 200;
```

Because the first half of the expression is `false`, the entire evaluation is `false`, and `bonus` is not set to 200. However, a side effect occurs with the `&` operator that does not occur when the `&&` operator is used: `yearsOfService` is incremented.

In general, you should avoid writing expressions that contain side effects. If you want `yearsOfService` to increase no matter what the value of `salesAmountForYear` is, then you should increase it in a stand-alone statement before the decision is made. If you want it increased only when the sales amount exceeds 10,000, then you should increase it in a statement that depends on that decision.



The `&` and `|` operators are Boolean logical operators when they are placed between Boolean expressions. When the same operators are used between integer expressions, they are **bitwise operators** that are used to manipulate the individual bits of values.

## Combining AND and OR Operators

You can combine as many AND and OR operators in an expression as you need. For example, when three conditions must be `true` before performing an action, you can use an expression such as `if(a && b && c)`. When you combine `&&` and `||` operators within the same Boolean expression, the `&&` operators take precedence, meaning their Boolean values are evaluated first.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

(continued)

2. Enter statements that describe the delivery charge criteria to the user and accept keyboard values for the customer's delivery zone and shipment size.

```
Console.WriteLine("Delivery is free for zone {0} or {1}",
    ZONE1, ZONE2);
Console.WriteLine("when the number of boxes is less than {0}",
    LOWQUANTITY);
Console.WriteLine("Enter delivery zone ");
inputString = Console.ReadLine();
deliveryZone = Convert.ToInt32(inputString);
Console.WriteLine
    ("Enter the number of boxes in the shipment");
inputString = Console.ReadLine();
quantity = Convert.ToInt32(inputString);
```

3. Write a compound if statement that tests whether the customer lives in Zone 1 or 2 and has a shipment consisting of fewer than 10 boxes. Notice that the first two comparisons joined with the || operator are contained in their own set of nested parentheses.

```
if((deliveryZone == ZONE1 || deliveryZone == ZONE2) &&
    quantity < LOWQUANTITY)
    Console.WriteLine("Delivery is free");
else
    Console.WriteLine("A delivery charge applies");
```

4. Add closing curly braces for the Main() method and for the class, and save the program. Compile and execute the program. Enter values for the zone and shipment size. Figure 4-19 shows the output.

```
C:\>C#\Chapter.04>DemoORAndAND
Delivery is free for zone 1 or 2
when the number of boxes is less than 10
Enter delivery zone
1
Enter the number of boxes in the shipment
20
A delivery charge applies

C:\>C#\Chapter.04>
```

**Figure 4-19** Sample execution of DemoORAndAND program

(continues)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
switch(year)
{
    case 1:
        Console.WriteLine("Freshman");
        break;
    case 2:
        Console.WriteLine("Sophomore");
        break;
    case 3: _____
    case 4:
        Console.WriteLine("Upperclass");
        break;
    default:
        Console.WriteLine("Invalid year");
        break;
}
```

Cases 3 and 4 are  
both "Upperclass".

**Figure 4-22** Example **switch** structure using multiple labels to execute a single statement block

Using a **switch** structure is never required; you can always achieve the same results with nested **if** statements. The **switch** statement is not as flexible as the **if** statement because you can test only one variable, and it must be tested for equality. The **switch** structure is simply a convenience you can use when there are several alternative courses of action depending on a match with a variable. Additionally, it makes sense to use a **switch** only when there are a reasonable number of specific matching values to be tested. For example, if every sale amount from \$1 to \$500 requires a 5 percent commission, it is not reasonable to test every possible dollar amount using the following code:

```
switch(saleAmount)
{
    case 1:
    case 2:
    case 3:
    // ...and so on for several hundred more cases
    commRate = .05;
    break;
```

With 500 different dollar values resulting in the same commission, one test—  
`if(saleAmount <= 500)`—is far more reasonable than listing 500 separate cases.

## Using an Enumeration with a **switch** Statement

Using an enumeration with a **switch** structure can often be convenient. Recall from Chapter 2 that an enumeration allows you to apply values to a list of constants. For example, Figure 4-23 shows a program that uses an enumeration to represent major courses of study at a college. In the enumeration list in Figure 4-23, ACCOUNTING is assigned 1, so the other values in the list are 2, 3, 4, and 5 in order. Suppose that students who are Accounting, CIS, or



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

## Using the NOT Operator

You use the **NOT operator**, which is written as an exclamation point (!), to negate the result of any Boolean expression. Any expression that evaluates as **true** becomes **false** when preceded by the ! operator, and any **false** expression preceded by the ! operator becomes **true**.



In Chapter 2 you learned that an exclamation point and equal sign together form the “not equal to” operator.

For example, suppose that a monthly car insurance premium is \$200 if the driver is younger than age 26 and \$125 if the driver is age 26 or older. Each of the following **if** statements (which have been placed on single lines for convenience) correctly assigns the premium values.

```
if(age < 26) premium = 200; else premium = 125;  
if(!(age < 26)) premium = 125; else premium = 200;  
if(age >= 26) premium = 125; else premium = 200;  
if(!(age>= 26)) premium = 200; else premium = 125;
```

The statements with the ! operator are somewhat more difficult to read, particularly because they require the double set of parentheses, but the result is the same in each case. Using the ! operator is clearer when the value of a Boolean variable is tested. For example, a variable initialized as `bool oldEnough = (age >= 25);` can become part of the relatively easy-to-read expression `if(!oldEnough)... .`

The ! operator has higher precedence than the **&&** and **||** operators. For example, suppose that you have declared two Boolean variables named `ageOverMinimum` and `ticketsUnderMinimum`. The following expressions are evaluated in the same way:

```
ageOverMinimum && !ticketsUnderMinimum  
ageOverMinimum && (!ticketsUnderMinimum)
```



Augustus de Morgan was a 19<sup>th</sup>-century mathematician who originally observed the following:

`!(a && b)` is equivalent to `!a || !b`  
`!(a || b)` is equivalent to `!(a && b)`



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Similarly, your boss might request, “Output the names of those employees in departments 1 and 2.” Because the boss used the word “and” in the request, you might be tempted to write the following:

```
if(department == 1 && department == 2) _____ This expression can never be true.  
    Console.WriteLine("Name is: {0}", name);
```

However, the variable `department` can never contain both a 1 and a 2 at the same time, so no employee name will ever be output, no matter what department the employee is in.

The correct statement is:

```
if(department == 1 || department == 2)  
    Console.WriteLine("Name is: {0}", name);
```

## Using the ! Operator Correctly

Whenever you use negatives, it is easy to make logical mistakes. For example, suppose that your boss says, “Make sure if the sales code is not ‘A’ or ‘B’, the customer gets a 10% discount.” You might be tempted to code the following:

```
if(salesCode != 'A' || salesCode != 'B') _____ This expression can never be true.  
    discount = 0.10;
```

However, this logic will result in every customer receiving the 10% discount because every `salesCode` is either not ‘A’ or not ‘B’. For example, if `salesCode` is ‘A’, then it is not ‘B’. The expression `salesCode != 'A' || salesCode != 'B'` is always `true`. The correct statement is either one of the following:

```
if(salesCode != 'A' && salesCode != 'B')  
    discount = 0.10;  
  
if(!(salesCode == 'A' || salesCode == 'B'))  
    discount = 0.10;
```

In the first example, if `salesCode` is not ‘A’ and it also is not ‘B’, then the discount is applied correctly. In the second example, if `salesCode` is ‘A’ or ‘B’, the inner Boolean expression is `true`, and the NOT operator (!) changes the evaluation to `false`, not applying the discount for ‘A’ or ‘B’ sales. You also could avoid the confusing negative situation by asking questions in a positive way, as in the following:

```
if(salesCode == 'A' || salesCode == 'B')  
    discount = 0;  
else  
    discount = 0.10;
```



Watch the video *Avoiding Common Decision Errors*.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



## You Do It

### *Creating a GUI Application that Uses an Enumeration and a switch Structure*

In these steps, you create a GUI application for the Chatterbox Diner that allows a user to enter a day and see the special meal offered that day. Creating the program provides experience using an enumeration in a switch structure.

1. Open a new project in Visual Studio, and name it **DailySpecial**.
2. Design a Form like the one in Figure 4-27 that prompts the user for a day number and allows the user to enter it in a TextBox. Name the TextBox **dayBox** and the Button **specialButton**.



**Figure 4-27** The Daily Special form

3. Below the Button, add a Label named **outputLabel** and delete its text.

(continues)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

3. What is the output of the following code segment?

```
int a = 3, b = 4;  
if(a > b)  
    Console.WriteLine("Black ");  
else  
    Console.WriteLine("White");
```

- a. Black
  - b. White
  - c. Black White
  - d. nothing

4. If the following code segment compiles correctly, what do you know about the variable `x`?

```
if(x) Console.WriteLine("OK");
```

- a.  $x$  is an integer variable
  - b.  $x$  is a Boolean variable
  - c.  $x$  is greater than 0
  - d. none of these

5. What is the output of the following code segment?

```
int c = 6, d = 12;  
if(c > d);  
    Console.WriteLine("Green ");  
    Console.WriteLine("Yellow");
```

- a. Green
  - b. Yellow
  - c. Green Yellow
  - d. nothing

6. What is the output of the following code segment?

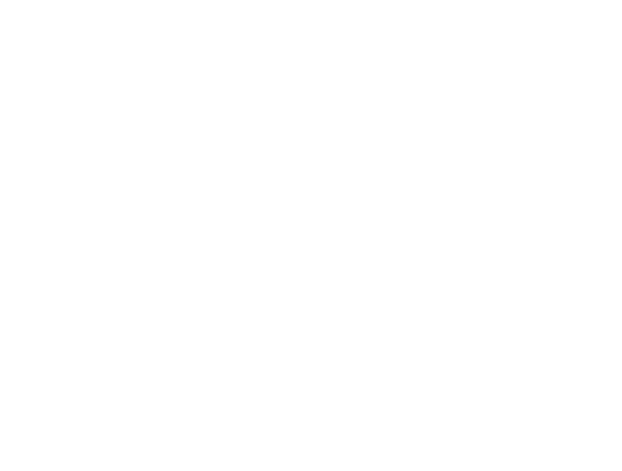
```
int c = 6, d = 12;
if(c < d)
    if(c > 8)
        Console.WriteLine("Green");
    else
        Console.WriteLine("Yellow");
else
    Console.WriteLine("Blue");
```

- a. Green
  - b. Yellow
  - c. Blue
  - d. nothing

7. What is the output of the following code segment?

```
int e = 5, f = 10;  
if(e < f && f < 0)  
    Console.WriteLine("Red ");  
else  
    Console.WriteLine("Orange");
```

- a. Red
  - b. Orange
  - c. Red Orange
  - d. nothing



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Write a console-based program named **GuessingGame** that generates a random number between 1 and 10. (In other words, `min` is 1 and `max` is 10.) Ask a user to guess the random number, then display the random number and a message indicating whether the user's guess was too high, too low, or correct.

- b. Create a GUI application named **GuessingGameGUI** that performs the same tasks as the program in Exercise 7a.
8. a. In the game Rock Paper Scissors, two players simultaneously choose one of three options: rock, paper, or scissors. If both players choose the same option, then the result is a tie. However, if they choose differently, the winner is determined as follows:
- Rock beats scissors, because a rock can break a pair of scissors.
  - Scissors beats paper, because scissors can cut paper.
  - Paper beats rock, because a piece of paper can cover a rock.
- Create a console-based game in which the computer randomly chooses rock, paper, or scissors. Let the user enter a character, 'r', 'p', or 's', each representing one of the three choices. Then, determine the winner. Save the application as **RockPaperScissors.cs**.
- b. Create a GUI application in which the user can play Rock, Paper, Scissors by clicking one of three buttons. Save the project as **RockPaperScissorsGUI**.
9. a. Create a console-based lottery game application named **Lottery**. Generate three random numbers, each between 1 and 4. Allow the user to guess three numbers. Compare each of the user's guesses to the three random numbers and display a message that includes the user's guess, the randomly determined three-digit number, and the amount of money the user has won as follows:

Matching Numbers	Award (\$)
Any one matching	10
Two matching	100
Three matching, not in order	1000
Three matching in exact order	10,000
No matches	0



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

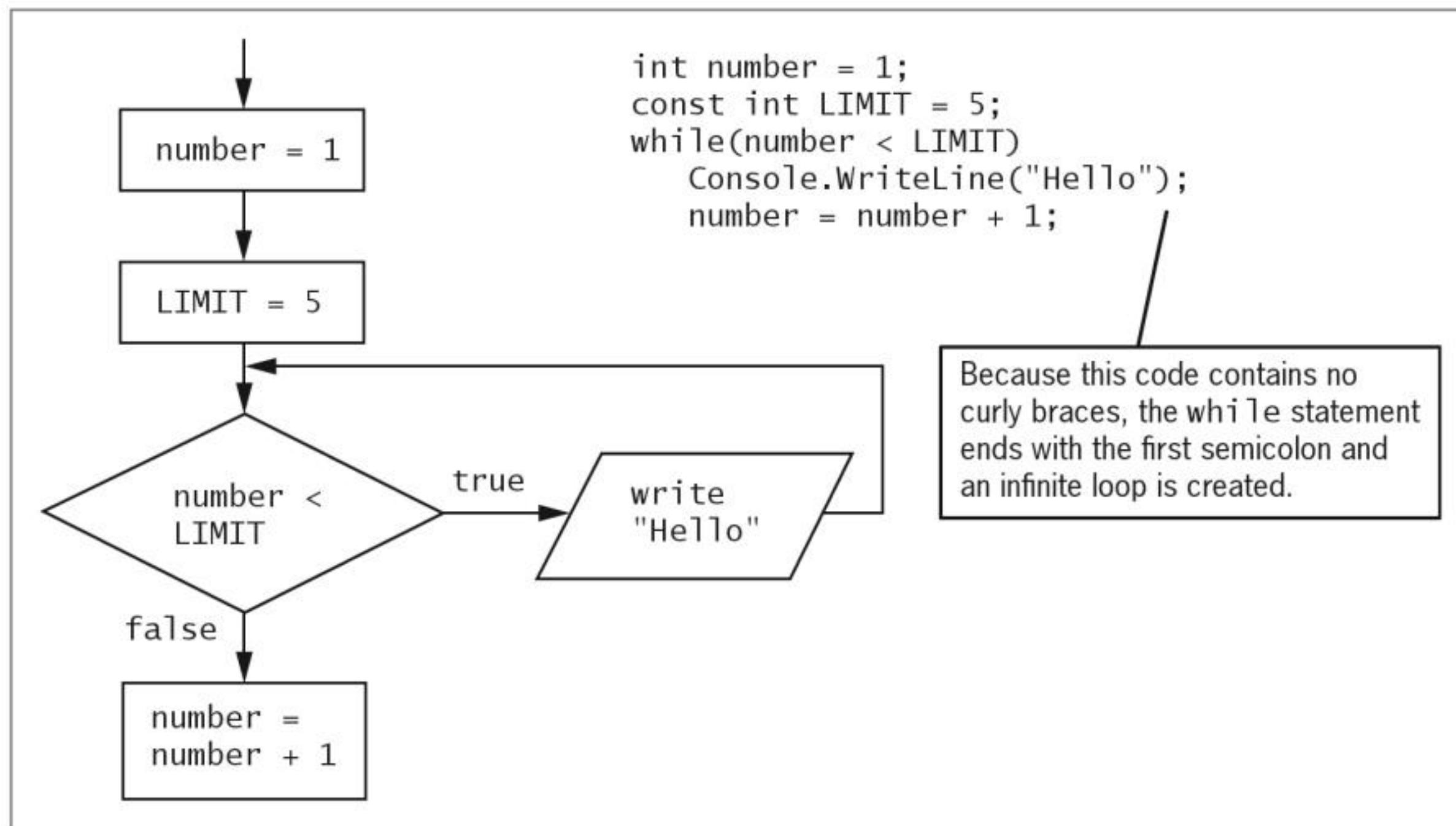


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The curly braces surrounding the body of the `while` loop in Figure 5-3 are important. If they are omitted, the `while` loop ends at the end of the “Hello” statement. Adding 1 to `number` would no longer be part of the loop body, so an infinite loop would be created. Even if the statement `number = number + 1;` was indented under the `while` statement, it would not be part of the loop without the surrounding curly braces. Figure 5-5 shows the incorrect logic that would result from omitting the curly braces. Many programmers recommend surrounding a loop body with curly braces even when there is only one statement in the body.



**Figure 5-5** Incorrect logic when curly braces are omitted from the loop in the `FourHello` program

Also, if a semicolon is mistakenly placed at the end of the partial statement, as in Figure 5-6, then the loop is also infinite. This loop has an **empty body**, or a body with no statements in it. In this case, `number` is initialized to 1, the Boolean expression `number < LIMIT` evaluates, and because it is `true`, the loop body is entered. Because the loop body is empty, ending at the semicolon, no action takes place, and the Boolean expression evaluates again. It is still `true` (nothing has changed), so the empty body is entered again, and the infinite loop continues. The program can never progress to either the statement that displays “Hello” or the statement that increases the value of `number`. The fact that these two statements are blocked using curly braces has no effect because of the incorrectly placed semicolon.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



## You Do It

### Using a `while` Loop

205

In the next steps, you write a program that continuously prompts the user for a valid ID number until the user enters one. For this application, assume that a valid ID number must be between 1000 and 9999 inclusive.

1. Open a new project named **ValidID**, and enter the beginning of the program by declaring variables for an ID number, the user's input, and constant values for the highest and lowest acceptable ID numbers.

```
using System;
class ValidID
{
    static void Main()
    {
        int idNum;
        string input;
        const int LOW = 1000;
        const int HIGH = 9999;
```

2. Add code to prompt the user for an ID number and to then convert it to an integer.

```
Console.WriteLine("Enter an ID number: ");
input = Console.ReadLine();
idNum = Convert.ToInt32(input);
```

3. Create a loop that continues while the entered ID number is out of range. While the number is invalid, explain valid ID parameters and reprompt the user, converting the input to an integer.

```
while(idNum < LOW || idNum > HIGH)
{
    Console.WriteLine("{0} is an invalid ID number", idNum);
    Console.Write("ID numbers must be ");
    Console.WriteLine("between {0} and {1} inclusive",
        LOW, HIGH);
    Console.WriteLine("Enter an ID number: ");
    input = Console.ReadLine();
    idNum = Convert.ToInt32(input);
}
```

(continues)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



## You Do It

210

### Using a *for* Loop

In the next steps, you write a program that creates a tipping table that restaurant patrons can use to approximate the correct tip for meals. Prices range from \$10 to \$100, and tipping percentage rates range from 10 percent to 25 percent. The program uses several loops.

1. Open a new file to start a program named **TippingTable**. It begins by declaring variables to use for the price of a dinner, a tip percentage rate, and the amount of the tip.

```
using System;
class TippingTable
{
    static void Main()
    {
        double dinnerPrice = 10.00;
        double tipRate;
        double tip;
```

2. Next, create some constants. Every tip from 10 percent through 25 percent will be computed in 5 percent intervals, so declare those values as **LOWRATE**, **MAXRATE**, and **TIPSTEP**. Tips will be calculated on dinner prices up to \$100.00 in \$10.00 intervals, so declare those constants too.

```
const double LOWRATE = 0.10;
const double MAXRATE = 0.25;
const double TIPSTEP = 0.05;
const double MAXDINNER = 100.00;
const double DINNERSTEP = 10.00;
```

3. To create a heading for the table, display “Price”. (For alignment, insert three spaces after the quotes and before the *P* in *Price*.) On the same line, use a loop that displays every tip rate from **LOWRATE** through **MAXRATE** in increments of **TIPSTEP**. In other words, the tip rates are 0.10, 0.15, 0.20, and 0.25. Complete the heading for the table using a **WriteLine()** statement that advances the cursor to the next line of output and a **WriteLine()** statement that displays a dashed line.

```
Console.WriteLine(" Price");
for(tipRate = LOWRATE; tipRate <= MAXRATE; tipRate +=
    TIPSTEP)
    Console.WriteLine("{0, 8}", tipRate.ToString("F"));
Console.WriteLine();
Console.WriteLine("-----");
```

(continues)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

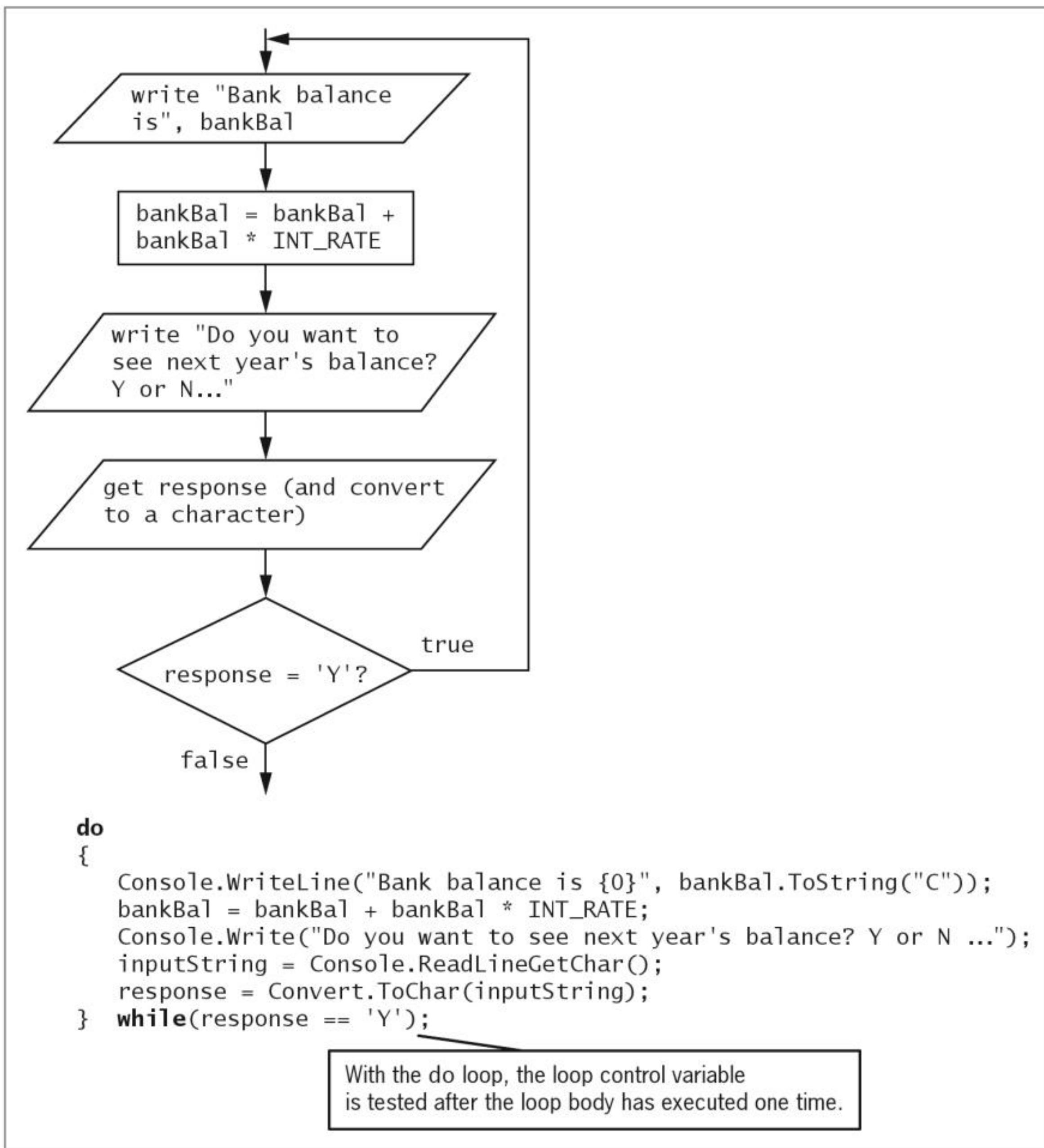


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

214



**Figure 5-14** Part of the bank balance program using a do loop



In a **do** loop, as a matter of style, many programmers prefer to align the **while** expression with the **do** keyword that starts the loop. Others feel that placing the **while** expression on its own line increases the chances that readers might misinterpret the line as the start of its own **while** statement instead of marking the end of a **do** statement.

In any situation where you want to loop, you never are required to use a **do** loop. Within the bank balance example, you could unconditionally display the bank balance once, prompt the user, and then start a **while** loop that might not be entered. However, when a task must be performed at least one time, the **do** loop is convenient.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

When you use a loop within a loop, you should always think of the outer loop as the all-encompassing loop. When you describe the task at hand, you often use the word “each” to refer to the inner loop. For example, if you wanted to output balances for different interest rates each year for 10 years, you could appropriately initialize some constants as follows:

218

```
const double RATE1 = 0.03;
const double RATE2 = 0.07
const double RATE_INCREASE = 0.01
const int END_YEAR = 10;
```

Then you could use the following nested `for` loops:

```
for(rate = RATE1; rate <= RATE2; rate += RATE_INCREASE)
    for(year = 1; year <= END_YEAR; ++year)
        Console.WriteLine(bankBal + bankBal * rate);
```

However, if you wanted to display balances for years 1 through 10 for each possible interest rate, you would use the following:

```
for(year = 1; year <= END_YEAR; ++year)
    for(rate = RATE1; rate <= RATE2; rate += RATE_INCREASE)
        Console.WriteLine(bankBal + bankBal * rate);
```

In both of these examples, the same 50 values would be displayed—five different interest rates for 10 years. However, in the first example, balances for years 1 through 10 would be displayed “within” each interest rate, and in the second example, each balance for each interest rate would be displayed “within” each year, 1 through 10. In other words, in the first example, the first 10 amounts displayed would be annual values using a rate of 0.03, and in the second example, the first five amounts displayed would be based on different interest values in the first year.



Watch the video *Using Nested Loops*.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

far more likely to enter a value that is too high than to enter a negative value, then you want to start a loop that reprompts the user with the following expression:

```
while(requestedNum > 15 || requestedNum < 1)...
```

Because you believe that the first Boolean expression is more likely to be true than the second one, you can eliminate testing the second one on more occasions. The order of the expressions is not very important in a single loop, but if this loop is nested within another loop, then the difference in the number of tests increases. Similarly, the order of the evaluations in `if` statements is more important when the `if` statements are nested within a loop.

## Comparing to Zero

Making a comparison to 0 is faster than making a comparison to any other value. Therefore, if your application makes comparison to 0 feasible, you can improve performance by structuring your loops to compare the loop control variable to 0 instead of some other value. For example, a loop based on a variable that ranges from 0 up to 100,000 executes the same number of times as a loop based on a variable that varies from 100,000 down to 0. However, the second loop performs slightly faster. Comparing a value to 0 instead of other values is faster because in a compiled language, condition flags for the comparison are set once, no matter how many times the loop executes. Comparing a value to 0 is faster than comparing to other values, no matter which comparison operator you use—greater than, less than, equal to, and so on.

You can prove that loops that use zero comparisons are faster by writing a test program. An easy way to compare loop execution times is to use C#'s `Stopwatch` class. You will better understand how this class works after you read Chapter 9, *Using Classes and Objects*, but you can use the class now by making statements similar to the following:

```
Stopwatch sw = Stopwatch.StartNew();
// Place statements to be timed here
sw.Stop();
```

The first statement creates and starts a `Stopwatch` object for timing events. The object's name is `sw`—you can use any legal C# identifier. The last statement stops the `Stopwatch`.

Subsequently, you can use a statement similar to the following to output the object's elapsed time in milliseconds:

```
Console.WriteLine("Time used: {0} ms", sw.Elapsed.TotalMilliseconds);
```

Figure 5-20 contains a program that uses the `Stopwatch` class to test the execution times of two do-nothing loops. (A **do-nothing loop** is one that performs no actions other than looping.) The statement `using System.Diagnostics;` is needed for the `Stopwatch` class. Both loops in the figure execute the same number of times, but one uses a zero comparison and the other does not. Before each loop, a `Stopwatch` object is started, and after each loop repeats 10 million times, the `Stopwatch` is stopped. As the execution in Figure 5-21 shows, there is a small difference in execution time between the two loops—about five milliseconds. The amount of time will vary on different machines, and it will also vary if you run the same



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

As you continue to study programming, you will discover many situations in which you can make your programs more efficient. You should always be on the lookout for ways to improve program performance without sacrificing readability.

### TWO TRUTHS & A LIE

227

#### Improving Loop Performance

1. You can improve loop performance by making sure the loop does not include unnecessary operations or statements.
2. You can improve loop performance by declaring temporary variables outside of a loop instead of continuously redeclaring them.
3. You can improve loop performance by omitting the initialization of the loop control variable.

The false statement is #3. A loop control variable must be initialized for every loop.

## Looping Issues in GUI Programs

Using a loop within a method in a GUI application is no different from using one in a console application; you can use `while`, `for`, and `do` statements in the same ways in both types of programs. For example, Figure 5-24 shows a GUI Form that prompts a user to enter a number and then displays “Hello” the corresponding number of times. The image on the left shows the Form when the program starts, and the image on the right shows the output after the user enters a value and clicks the button. Figure 5-25 shows the code in the `greetingsButton_Click()` method. When a user clicks the `greetingsButton`, an integer is extracted from the `TextBox` on the Form. Then a `for` loop appends “Hello” and a newline character to the `Text` property of the `outputLabel` the correct number of times.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

7. What is the major advantage of using a **for** loop instead of a **while** loop?
- With a **for** loop, it is impossible to create an infinite loop.
  - It is the only way to achieve an indefinite loop.
  - Unlike with a **while** loop, the execution of multiple statements can depend on the test condition.
  - The loop control variable is initialized, tested, and altered all in one place.
8. A **for** loop statement must contain \_\_\_\_\_.
- |                   |               |
|-------------------|---------------|
| a. two semicolons | c. four dots  |
| b. three commas   | d. five pipes |
9. In a **for** statement, the section before the first semicolon executes \_\_\_\_\_.
- once
  - once prior to each loop iteration
  - once after each loop iteration
  - one less time than the initial loop control variable value
10. The three sections of the **for** loop are most commonly used for \_\_\_\_\_ the loop control variable.
- testing, outputting, and incrementing
  - initializing, testing, and incrementing
  - incrementing, selecting, and testing
  - initializing, converting, and outputting
11. Which loop is most convenient to use if the loop body must always execute at least once?
- |                        |                      |
|------------------------|----------------------|
| a. a <b>do</b> loop    | c. a <b>for</b> loop |
| b. a <b>while</b> loop | d. an <b>if</b> loop |
12. The loop control variable is checked at the bottom of which kind of loop?
- |                        |                      |
|------------------------|----------------------|
| a. a <b>while</b> loop | c. a <b>for</b> loop |
| b. a <b>do</b> loop    | d. all of the above  |
13. A **for** loop is an example of a(n) \_\_\_\_\_ loop.
- |             |             |
|-------------|-------------|
| a. untested | c. posttest |
| b. pretest  | d. infinite |
14. A **while** loop is an example of a(n) \_\_\_\_\_ loop.
- |             |             |
|-------------|-------------|
| a. untested | c. posttest |
| b. pretest  | d. infinite |



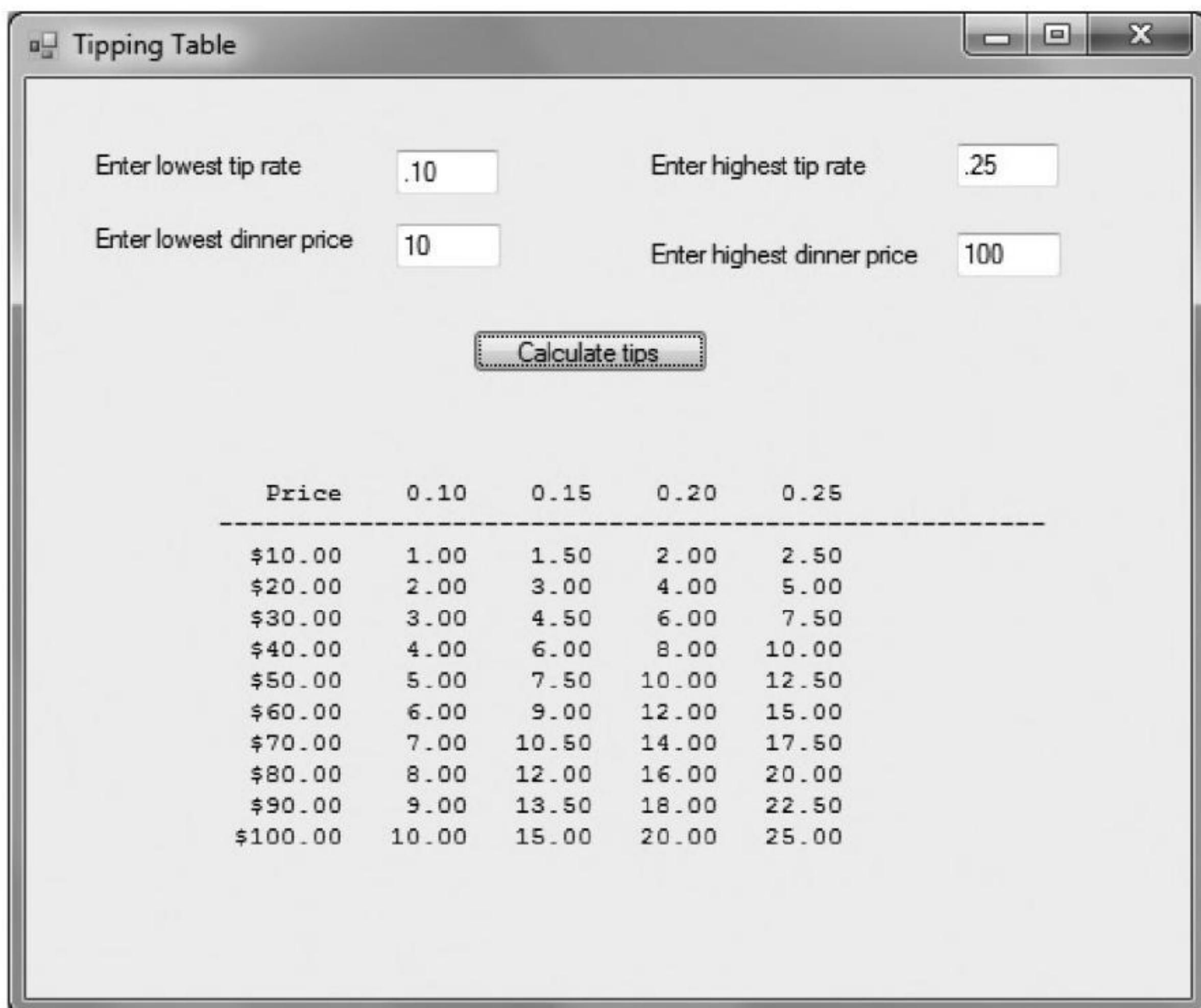
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



237

**Figure 5-28** Typical execution of `TippingTable2GUI` program

11. a. Write a console-based program named **WebAddress** that asks a user for a business name. Suggest a good Web address by adding “www.” to the front of the name, removing all spaces from the name, and adding “.com” to the end of the name. For example, a good Web address for “Acme Plumbing and Supply” is *www.AcmePlumbingandSupply.com*.  
b. Create a GUI application named **WebAddressGUI** that creates a Web address name from a business name, as described in Exercise 11a.
12. a. Write a console-based program named **CountVowels** that accepts a phrase from the user and counts the number of vowels in the phrase. For this exercise, count both uppercase and lowercase vowels, but do not consider “y” to be a vowel.  
b. Create a GUI application named **CountVowelsGUI** that counts vowels, as described in Exercise 12a.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Storing values in variables provides programs with flexibility; a program that uses variables to replace constants can manipulate different values each time it executes. When you add loops to your programs, the same variable can hold different values during successive cycles through the loop within the same program execution. This ability makes the program even more flexible. Learning to use the data structure known as an array offers further flexibility. Arrays allow you to store multiple values in adjacent memory locations and access them by varying a value that indicates which of the stored values to use. In this chapter, you will learn to create and manage C# arrays.

## Declaring an Array and Assigning Values to Array Elements

Sometimes, storing just one value in memory at a time isn't adequate. For example, a sales manager who supervises 20 employees might want to determine whether each employee has produced sales above or below the average amount. When you enter the first employee's sales figure into a program, you can't determine whether it is above or below average because you won't know the average until you have entered all 20 figures. You might plan to assign 20 sales figures to 20 separate variables, each with a unique name, then sum and average them. However, that process is awkward and unwieldy: You need 20 prompts, 20 input statements using 20 separate storage locations (in other words, 20 separate variable names), and 20 addition statements. This method might work for 20 salespeople, but what if you have 30, 40, or 10,000 salespeople?

A superior approach is to assign the sales value to the same variable in 20 successive iterations through a loop that contains one prompt, one input statement, and one addition statement. Unfortunately, when you enter the sales value for the second employee, that data item replaces the figure for the first employee, and the first employee's value is no longer available to compare to the average of all 20 values. With this approach, when the data-entry loop finishes, the only sales value left in memory is the last one entered.

The best solution to this problem is to create an array. An **array** is a list of data items that all have the same data type and the same name. Each object in an array is an **array element**. You can distinguish each element from the others in an array with a subscript. A **subscript** (also called an **index**) is an integer contained within square brackets that indicates the position of particular array elements.

You declare an array variable with a data type, a pair of square brackets, and an identifier. For example, to declare an array of **double** values to hold sales figures for salespeople, you write the following:

```
double[] sales;
```



In some programming languages, such as C++ and Java, you also can declare an array variable by placing the square brackets after the array name, as in `double sales[];`. This format is illegal in C#.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

To increase each array element by 3, for example, you can write the following five statements:

```
myScores[0] += 3;  
myScores[1] += 3;  
myScores[2] += 3;  
myScores[3] += 3;  
myScores[4] += 3;
```

You can shorten the task by using a variable as a subscript. Then you can use a loop to perform arithmetic on each element in the array. For example, you can use a `while` loop, as follows:

```
int sub = 0;  
while(sub < 5)  
{  
    myScores[sub] += 3;  
    ++sub;  
}
```

You also can use a `for` loop, as follows:

```
for(int sub = 0; sub < 5; ++sub)  
    myScores[sub] += 3;
```

In both examples, the variable `sub` is declared and initialized to 0, then compared to 5. Because it is less than 5, the loop executes and `myScores[0]` increases by 3. The variable `sub` is incremented and becomes 1, which is still less than 5, so when the loop executes again, `myScores[1]` increases by 3, and so on. If the array had 100 elements, individually increasing the array values by 3 would require 95 additional statements, but the only change required using either loop would be to change the limiting value for `sub` from 5 to 100.

New array users sometimes think there is a permanent connection between a variable used as a subscript and the array with which it is used, but that is not the case. For example, if you vary `sub` from 0 to 10 to fill an array, you do not need to use `sub` later when displaying the array elements—either the same variable or a different variable can be used as a subscript elsewhere in the program.

## Using the Length Property

When you work with array elements, you must ensure that the subscript you use remains in the range of 0 through one less than the array's length. If you declare an array with five elements and use a subscript that is negative or more than 4, you will receive the error message “`IndexOutOfRangeException`” when you run the program. (You will learn about the `IndexOutOfRangeException` in the chapter *Exception Handling*.) This message means the index, or subscript, does not hold a value that legally can access an array element. For example, if you declare an array of five integers, you can display them as follows:

```
int[] myScores = {100, 75, 88, 100, 90};  
for(int sub = 0; sub < 5; ++sub)  
    Console.WriteLine("{0} ", myScores[sub]);
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
if(itemOrdered == 101)
    isValidItem = true;
else if(itemOrdered == 108)
    isValidItem = true;
else if(itemOrdered == 201)
    isValidItem = true;
// and so on
```

252

Instead of creating a long series of `if` statements, a more elegant solution to determining whether a value is valid is to compare it to a list of values in an array. For example, you can initialize an array with the valid values by using the following statement:

```
int[] validValues = {101, 108, 201, 213, 266, 304, 311,
    409, 411, 412};
```



You might prefer to declare the `validValues` array as a constant because, presumably, the valid item numbers should not change during program execution. In C# you must use the keywords `static` and `readonly` prior to the constant declaration. To keep these examples simple, all arrays in this chapter are declared as variable arrays.

After the `validValues` array is declared, you can use either a `for` loop or a `while` loop to search whether the `itemOrdered` variable value matches any of the array entries.

## Using a for Loop to Search an Array

One way to determine whether an `itemOrdered` value equals a value in the `validValues` array is to use a `for` statement to loop through the array and set a Boolean variable to `true` when a match is found:

```
for(int x = 0; x < validValues.Length; ++x)
    if(itemOrdered == validValues[x])
        isValidItem = true;
```

This type of search is called a **sequential search** because each array element is examined in sequence. This simple `for` loop replaces the long series of `if` statements. What's more, if a company carries 1000 items instead of 10, then the list of valid items in the array must be altered, but the `for` statement does not change at all. As an added bonus, if you set up another array as a **parallel array** with the same number of elements and corresponding data, you can use the same subscript to access additional information. For example, if the 10 items your company carries have 10 different prices, then you can set up an array to hold those prices as follows:

```
double[] prices = {0.89, 1.23, 3.50, 0.69...}; // and so on
```

The prices must appear in the same order as their corresponding item numbers in the `validValues` array. Now the same `for` loop that finds the valid item number also finds the price, as shown in the program in Figure 6-5. In other words, if the item number is found in the second position in the `validValues` array, then you can find the correct price in the second position in the `prices` array. In the program in Figure 6-5, the variable used as a subscript, `x`, is set to 0 and the Boolean variable `isValidItem` is `false`. In the shaded portion



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
using System;
class FindPriceWithWhileLoop
{
    static void Main()
    {
        int x;
        string inputString;
        int itemOrdered;
        double itemPrice = 0;
        bool isValidItem = false;
        int[] validValues = {101, 108, 201, 213, 266,
            304, 311, 409, 411, 412};
        double[] prices = {0.89, 1.23, 3.50, 0.69, 5.79,
            3.19, 0.99, 0.89, 1.26, 8.00};
        Console.WriteLine("Enter item number ");
        inputString = Console.ReadLine();
        itemOrdered = Convert.ToInt32(inputString);
        x = 0;
        while(x < validValues.Length &&
            itemOrdered != validValues[x])
            ++x;
        if(x != validValues.Length)
        {
            isValidItem = true;
            itemPrice = prices[x];
        }
        if(isValidItem)
            Console.WriteLine("Item {0} sells for {1}",
                itemOrdered, itemPrice.ToString("C"));
        else
            Console.WriteLine("No such item as {0}",
                itemOrdered);
    }
}
```

**Figure 6-8** The FindPriceWithWhileLoop program that searches with a `while` loop

In the application in Figure 6-8, the variable used as a subscript, `x`, is set to 0 and the Boolean variable `isValidItem` is `false`. In the shaded portion of the figure, while the subscript remains smaller than the length of the array of valid item numbers, and while the user's requested item does not match a valid item, the subscript is increased so that subsequent array values can be tested. The `while` loop ends when a match is found or the array tests have been exhausted, whichever comes first. When the loop ends, if `x` is not equal to the size of the array, then a valid item has been found and its price can be retrieved from the `prices` array. Figure 6-9 shows two executions of the program. In the first execution, a match is found; in the second, an invalid item number is entered, so no match is found.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



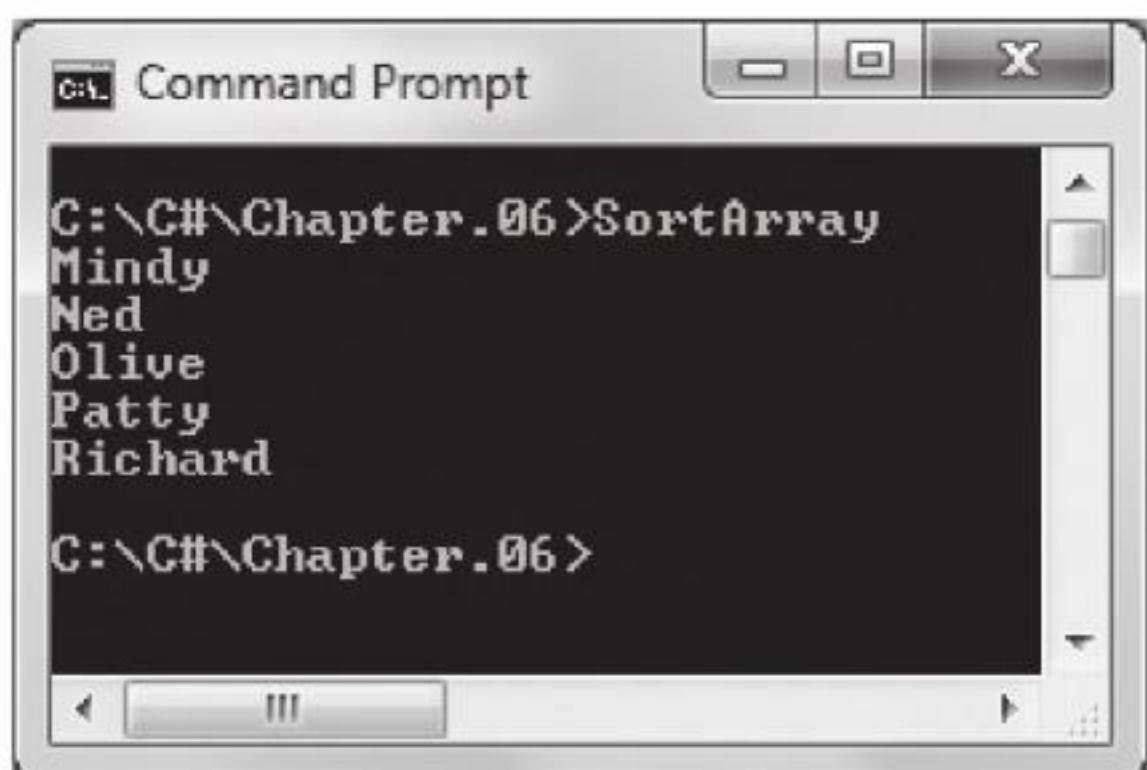
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

To use the method, you pass the array name to `Array.Sort()`, and the element positions within the array are rearranged appropriately. Figure 6-14 shows a program that sorts an array of strings; Figure 6-15 shows its execution.

262

```
using System;
class SortArray
{
    static void Main()
    {
        string[] names = {"Olive", "Patty",
                          "Richard", "Ned", "Mindy"};
        int x;
        Array.Sort(names);
        for(x = 0; x < names.Length; ++x)
            Console.WriteLine(names[x]);
    }
}
```

**Figure 6-14** SortArray program



**Figure 6-15** Execution of SortArray program

Because the `BinarySearch()` method requires that array elements be sorted in order, the `Sort()` method is often used in conjunction with it.



The `Array.Sort()` method provides a good example of encapsulation—you can use the method without understanding how it works internally. The method actually uses an algorithm named Quicksort. You will learn how to implement this algorithm yourself as you continue to study programming.

## Using the `Reverse()` Method

The **Reverse() method** reverses the order of items in an array. In other words, for any array, the element that starts in position 0 is relocated to position `Length - 1`, the element that starts in position 1 is relocated to position `Length - 2`, and so on until the element that starts



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

<code>sales[0, 0]</code>	<code>sales[0, 1]</code>	<code>sales[0, 2]</code>	<code>sales[0, 3]</code>
<code>sales[1, 0]</code>	<code>sales[1, 1]</code>	<code>sales[1, 2]</code>	<code>sales[1, 3]</code>
<code>sales[2, 0]</code>	<code>sales[2, 1]</code>	<code>sales[2, 2]</code>	<code>sales[2, 3]</code>

**Figure 6-19** View of a rectangular, two-dimensional array in memory

The array in Figure 6-19 is a rectangular array. In a **rectangular array**, each row has the same number of columns. You must use two subscripts when you access an element in a two-dimensional array. When mathematicians use a two-dimensional array, they often call it a **matrix** or a **table**; you might have used a two-dimensional array called a spreadsheet. You might want to create a `sales` array with two dimensions as shown in Figure 6-19 if, for example, each row represented a category of items sold, and each column represented a salesperson who sold them.

When you declare a one-dimensional array, you type a single, empty set of square brackets after the array type, and you use a single subscript in a set of square brackets when reserving memory. To declare a two-dimensional array, you type a comma in the square brackets after the array type, and you use two subscripts, separated by a comma in brackets, when reserving memory. For example, the array in Figure 6-19 can be declared as the following, creating an array named `sales` that holds three rows and four columns:

```
double[ , ] sales = new double[3, 4];
```

When you declare a two-dimensional array, spaces surrounding the comma within the square brackets are optional.

Just as with a one-dimensional array, if you do not provide values for the elements in a two-dimensional numerical array, the values are set to the default value for the data type (for example, zero for numeric data). You can assign other values to the array elements later. For example, the following statement assigns the value 14.00 to the element of the `sales` array that is in the first column of the first row:

```
sales[0, 0] = 14.00;
```

Alternatively, you can initialize a two-dimensional array by assigning values when it is created. For example, the following code assigns values to `sales` upon declaration:

```
double[ , ] sales = {{14.00, 15.00, 16.00, 17.00},  
                      {21.99, 34.55, 67.88, 31.99},  
                      {12.03, 55.55, 32.89, 1.17}};
```

The `sales` array contains three rows and four columns. You contain the entire set of values within a pair of curly braces. The first row of the array holds the four `doubles` 14.00, 15.00, 16.00 and 17.00. Notice that these four values are placed within their own inner set of curly braces to indicate that they constitute one row, or the first row, which is row 0. The row and its curly braces are separated from the next row with a comma. The next four values in their



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

17. Which of the following traits do the `BinarySearch()` and `Sort()` methods have in common?
  - a. Both methods take a single argument that must be an array.
  - b. Both methods belong to the `System.Array` class.
  - c. The array that each method uses must be in ascending order.
  - d. They both operate on arrays made up of simple data types but not class objects.
18. If you use the `BinarySearch()` method and the object you seek is not found in the array, \_\_\_\_\_.
  - a. an error message is displayed
  - b. a zero is returned
  - c. the value `false` is returned
  - d. a negative value is returned
19. The `BinarySearch()` method is inadequate when \_\_\_\_\_.
  - a. array items are in ascending order
  - b. the array holds duplicate values and you want to find them all
  - c. you want to find an exact match for a value
  - d. array items are not numeric
20. Which of the following declares an integer array that contains eight rows and five columns?
  - a. `int[8, 5] num = new int[ , ];`
  - b. `int [8][5] num = new int[];`
  - c. `int [ , ] num = new int[5, 8];`
  - d. `int [ , ] num = new int[8, 5];`

## Exercises



### Programming Exercises

1. Write a program named **IntArrayDemo** that stores an array of 10 integers. Until the user enters a sentinel value, allow the user three options: (1) to view the list in order from the first to last position, (2) to view the list in order from the last to first position, or (3) to choose a specific position to view.
2. Write a program named **TipsList** that accepts seven `double` values representing tips earned by a waiter each day during the week. Display each of the values along with a message that indicates how far it is from the average.
3. Write a program named **ScoresComparison** that allows a user to input four integer quiz scores ranging from 0 through 100. If no score is lower than any



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

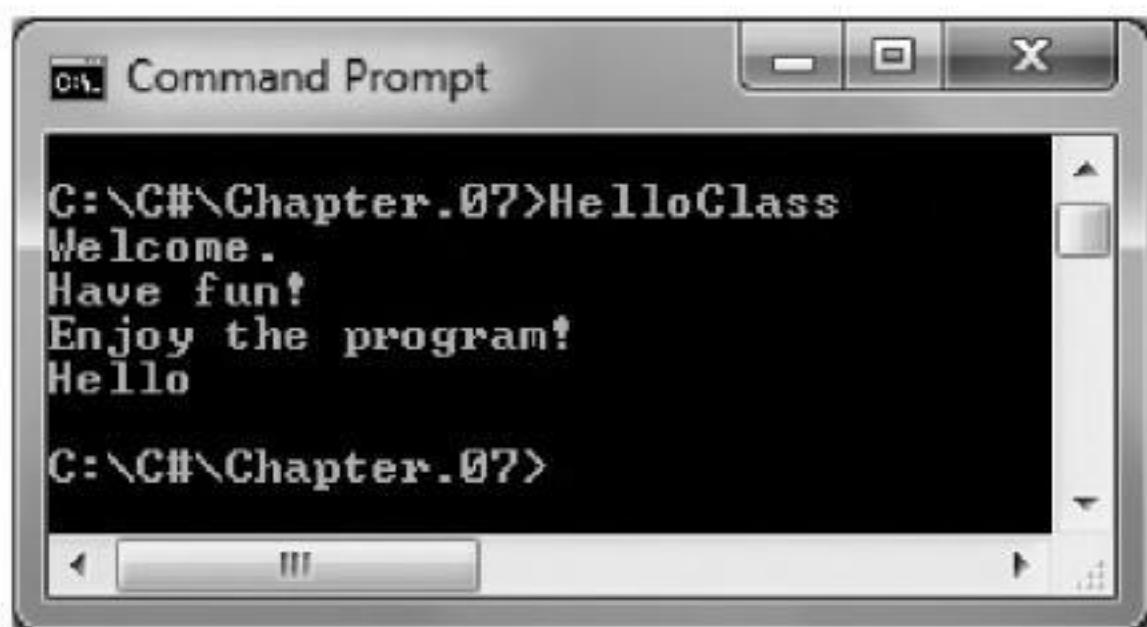


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
using System;
class HelloClass
{
    static void Main()
    {
        ShowWelcomeMessage();
        Console.WriteLine("Hello");
    }
    private static void ShowWelcomeMessage()
    {
        Console.WriteLine("Welcome.");
        Console.WriteLine("Have fun!");
        Console.WriteLine("Enjoy the program!");
    }
}
```

**Figure 7-4** HelloClass program with Main() method calling the ShowWelcomeMessage() method

The ShowWelcomeMessage() method in the HelloClass class is **static**, and therefore it can be called without creating an object from the Main() method, which is also **static**.



**Figure 7-5** Output of HelloClass program

When the Main() method executes, it calls the ShowWelcomeMessage() method, so the three lines that make up the welcome message appear first in the output in Figure 7-5. Then, after the method is done, the Main() method displays *Hello*.

Each of two different classes can have its own method named ShowWelcomeMessage(). Such a method in the second class would be entirely distinct from the identically named method in the first class. The complete name of this method is HelloClass.ShowWelcomeMessage(), but you do not need to use the complete name when calling the method within the class.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

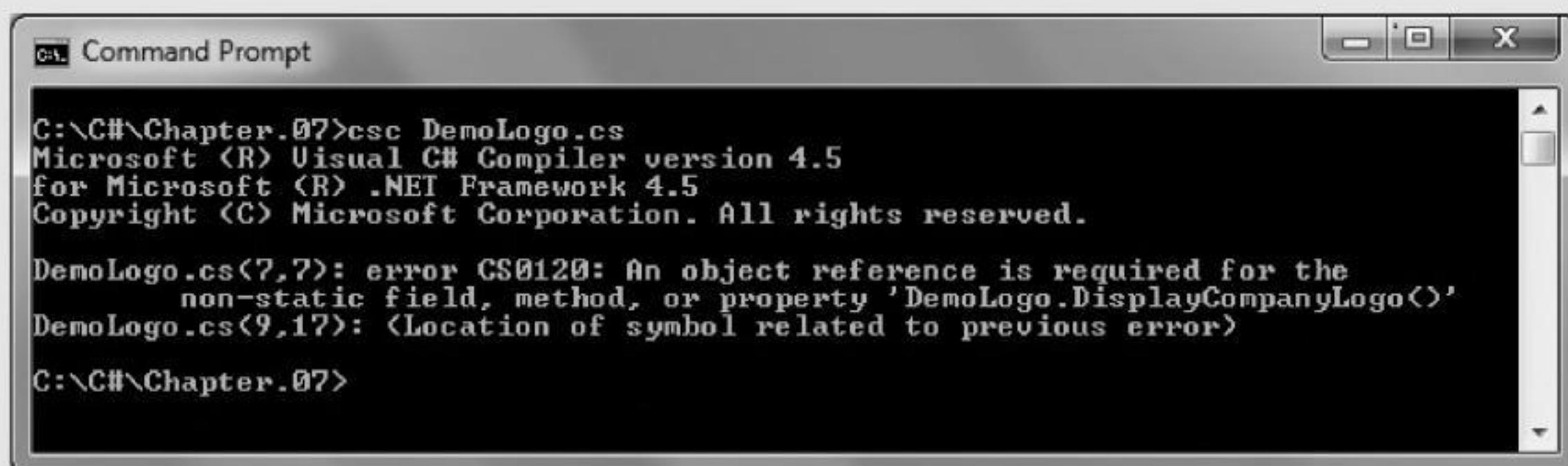


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

(continued)

6. Remove the keyword `static` from the `DisplayCompanyLogo()` method, and then save and compile the program. An error message appears, as shown in Figure 7-9. The message indicates that an object reference is required for the nonstatic `DisplayCompanyLogo()` method. The error occurs because the `Main()` method is `static`; therefore, without declaring an object, it can only call other `static` methods. Remember that you will learn to create objects in Chapter 9, and then you can create the types of methods that do not use the keyword `static`. For now, replace the keyword `static` in the `DisplayCompanyLogo()` method header, and compile and execute the program again.

295



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The command entered is `C:\>csc DemoLogo.cs`. The output shows the Microsoft Visual C# Compiler version 4.5 and a copyright notice. It then displays an error message: `DemoLogo.cs(7,7): error CS0120: An object reference is required for the non-static field, method, or property 'DemoLogo.DisplayCompanyLogo()'` and `DemoLogo.cs(9,17): <Location of symbol related to previous error>`. The prompt at the bottom is `C:\>C#\Chapter.07>`.

**Figure 7-9** Error message when calling a nonstatic method from a static method

7. Remove the keyword `private` from the header of the `DisplayCompanyLogo()` method. Save and compile the program, and then execute it. The execution is successful because the `private` keyword is optional. Replace the `private` keyword, and save the program.

## Writing Methods That Require a Single Argument

Some methods require additional information. If a method could not receive arguments, then you would have to write an infinite number of methods to cover every possible situation. For example, when you make a dental appointment, you do not need to employ a different method for every date of the year at every possible time of day. Rather, you can supply the date and time as information to the method, and no matter what date and time you supply, the method is carried out correctly. If you design a method to compute an employee's



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

## Writing Methods That Require Multiple Arguments

A method can require more than one argument. You can pass multiple arguments to a method by listing the arguments within the call to the method and separating them with commas. For example, rather than creating a `DisplaySalesTax()` method that multiplies an amount by 0.07, you might prefer to create a more flexible method to which you can pass two values—the value on which the tax is calculated and the tax percentage by which it should be multiplied. Figure 7-13 shows a method that uses two such arguments.

```
private static void DisplaySalesTax(double saleAmount, double taxRate)
{
    double tax;
    tax = saleAmount * taxRate;
    Console.WriteLine("The tax on {0} at {1} is {2}",
        saleAmount.ToString("C"),
        taxRate.ToString("P"), tax.ToString("C"));
}
```

**Figure 7-13** The `DisplaySalesTax()` method that takes two arguments

In Figure 7-13, two parameters (`saleAmount` and `taxRate`) appear within the parentheses in the method header. A comma separates the parameters, and each parameter requires its own named type (in this case, both parameters are of type `double`) and an identifier. A declaration for a method that receives two or more arguments must list the type for each parameter separately, even if the parameters have the *same* type.

When you pass values to the method in a statement such as `DisplaySalesTax(myPurchase, localRate);`, the first value passed will be referenced as `saleAmount` within the method, and the second value passed will be referenced as `taxRate`. Therefore, it is very important that arguments be passed to a method in the correct order. The following call results in output stating that “The tax on \$200.00 at 10.00% is \$20.00”:

```
DisplaySalesTax(200.00, 0.10);
```

However, the following call results in output stating that “The tax on \$0.10 at 10.00% is \$20.00”, which is clearly incorrect.

```
DisplaySalesTax(0.10, 200.00);
```



If two method parameters are of the same type—for example, two `doubles`—passing arguments to a method in the wrong order results in a logical error. For example, if you pass values to `DisplaySalesTax()` in the wrong order, the multiplication works but the output description is incorrect. However, if a method expects parameters of diverse types, then passing arguments in reverse order constitutes a syntax error, and the program will not compile.

You can write a method to take any number of parameters in any order. When you call the method, however, the arguments you send to it must match (in both number and type) the parameters listed in the method declaration, with the following exceptions:



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



## You Do It

306

### *Writing a Method that Receives Parameters and Returns a Value*

Next, you write a method named `CalcPhoneCallPrice()` that both receives parameters and returns a value. The purpose of the method is to accept the length of a phone call in minutes and the rate charged per minute, and to then calculate the price of a call, assuming each call includes a 25-cent connection charge in addition to the per-minute charge. After writing the `CalcPhoneCallPrice()` method, you write a `Main()` method that calls the `CalcPhoneCallPrice()` method using four different sets of data as arguments.

1. Start a program named `PhoneCall` by typing a `using` statement, class header, and opening brace:

```
using System;
class PhoneCall
{
```

2. Type the following `CalcPhoneCallPrice()` method. The method is declared as `static` because it will be called by a `static Main()` method without creating an object. The method receives an `integer` and a `double` as parameters. The fee for a call is calculated as 0.25 plus the minutes times the rate per minute. The method returns the phone call fee to the calling method.

```
private static double CalcPhoneCallPrice(int minutes,
    double rate)
{
    const double BASE_FEE = 0.25;
    double callFee;
    callFee = BASE_FEE + minutes * rate;
    return callFee;
}
```

3. Add the `Main()` method header for the `PhoneCall` class. Begin the method by declaring two arrays; one contains two call lengths and the other contains two rates. You will use all the possible combinations of call lengths and rates to test the `CalcPhoneCallPrice()` method. Also, declare a `double` named `priceOfCall` that will hold the result of a calculated call price.

```
static void Main()
{
    int[] callLengths = {2, 5};
    double[] rates = { 0.03, 0.12};
    double priceOfCall;
```

(continues)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The phrase `string[] args` is a parameter to the `Main()` method. The variable `args` represents an array of strings that you can pass to `Main()`. Although you can use any identifier, `args` is conventional. In particular, Java programmers might prefer the C# version of `Main()` that includes the `string[] args` parameter because their convention is to write main methods with the same parameter.

Use this format for the `Main()` method header if you need to access command-line arguments passed in to your application. For example, the program in Figure 7-22 displays an `args` array that is a parameter to its `Main()` method. Figure 7-23 shows how a program might be executed from the command line using arguments to `Main()`.

```
using System;
class DisplayArgs
{
    static void Main(string[] args)
    {
        for(int x = 0; x < args.Length; ++x)
            Console.WriteLine("Argument {0} is {1}", x, args[x]);
    }
}
```

**Figure 7-22** A `Main()` method with a `string[] args` parameter



**Figure 7-23** Executing the `DisplayArgs` program with arguments

Even if you do not need access to command-line arguments, you can still use the version of the `Main()` method header that references them. You should use this version if your instructor or supervisor indicates you should follow this convention.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- The return type for a method defines the type of value sent back to the calling method. It can be any type used in the C# programming language, which includes the basic built-in types `int`, `double`, `char`, and so on, as well as class types (including class types you create). A method also can return nothing, in which case the return type is `void`.
- You can pass an array as a parameter to a method. You indicate that a method parameter is an array by placing square brackets after the data type in the method's parameter list. Arrays, like all objects but unlike built-in types, are passed by reference; that is, the method receives the actual memory address of the array and has access to the actual values in the array elements.
- You might see different `Main()` method headers in other books or in programs written by others. Some programmers include an array of strings as a parameter to `Main()`, some programmers return an `int` from the `Main()` method, and some programmers give `Main()` methods public access.
- Special considerations exist for methods when you create GUI applications. You must understand the automatically generated methods in the visual environment, appreciate the differences in scope in GUI programs, and understand that many methods must be nonstatic when associated with a `Form`.

## Key Terms

A **method** is an encapsulated series of statements that carry out a task.

Methods are **invoked**, or **called**, by other methods.

A **calling method** calls another method.

A **called method** is invoked by another method.

**Implementation hiding** means keeping the details of a method's operations hidden.

To **interface** with a system is to interact with it.

A **client** is a method that uses another method.

A **black box** is any device you can use without knowing how it works internally.

A **multifile assembly** is a group of files containing methods that work together to create an application.

**Code bloat** is a term that describes unnecessarily long or repetitive program statements.

A **method declaration** is a **method header** or **method definition**.

A **method body** is a block of statements that carry out a method's work.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

5. a. Create a console-based program named **FormLetter** whose `Main()` method contains an array of at least six strings with sentences that encourage sales, such as “Buy it today” and “You won’t regret your purchase.” The program randomly selects three of the sentences and passes them to a method that generates a customer sales letter. The letter contains a greeting followed by the selected phrases.  
b. Create an application named **FormLetterGUI** that is similar to the application described in Exercise 5a, except that the letter is composed when the user clicks a button.
6. a. Create a console-based application named **Multiplication** whose `Main()` method asks the user to input an integer and then calls a method named `MultiplicationTable()`, which displays the results of multiplying the integer by each of the numbers 2 through 10.  
b. Create a GUI application named **MultiplicationGUI** that calls the `MultiplicationTable()` method described in Exercise 6a after the user enters an integer into a `TextBox` and clicks a `Button`.
7. In Chapter 4, you wrote a program named **Admission** for a college admissions office in which the user enters a numeric high school grade point average and an admission test score. The program displays “Accept” or “Reject” based on those values. Now, create a modified program named **AdmissionModularized** in which the grade point average and test score are passed to a method that returns a string containing “Accept” or “Reject”.
8. In Chapter 5, you wrote a program named **CountVowels** that accepts a phrase from the user and counts the number of vowels in the phrase. Now, create a modified program named **CountVowelsModularized** in which the phrase is passed to a method that returns the number of vowels.
9. In Chapter 6, you wrote an application named **Commission** that computes commissions for automobile salespeople based on the value of the car sold. Now, create a modified application named **CommissionModularized** in which a method accepts the car price and returns the commission rate.
10. a. Create a console-based application named **Desks** that computes the price of a desk and whose `Main()` method calls the following methods:
  - A method to accept the number of drawers in the desk as input from the keyboard. This method returns the number of drawers to the `Main()` method.
  - A method to accept as input and return the type of wood—‘m’ for mahogany, ‘o’ for oak, or ‘p’ for pine.
  - A method that accepts the number of drawers and wood type, and calculates the cost of the desk based on the following:



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

In the last chapter you learned to call methods, pass arguments to them, and receive values from them. In this chapter you will expand your method-handling skills to include more sophisticated techniques, including using reference, output, and optional parameters and overloading methods. Understanding how to manipulate methods is crucial when you are working on large, professional, real-world projects. With methods, you can coordinate your work with that of other programmers.

## Understanding Parameter Types

In C#, you can write methods with several kinds of formal parameters listed within the parentheses in the method header. Parameters can be mandatory or optional. When you use a **mandatory parameter**, an argument for it is required in every method call.

The four types of mandatory parameters are:

- Value parameters, when they are declared *without* default values
- Reference parameters, which are declared with the `ref` modifier
- Output parameters, which are declared with the `out` modifier
- Parameter arrays, which are declared with the `params` modifier

C# includes only one type of optional parameter:

- Value parameters, when they are declared *with* default values

## Using Mandatory Value Parameters

So far, all of the method parameters you have created have been mandatory, and all except arrays have been value parameters. When you use a **value parameter** in a method header, you indicate the parameter's type and name, and the method receives a copy of the value passed to it. A variable that is used as an argument to a method with a value parameter must have a value assigned to it. If it does not, the program will not compile.

The value of a method's value parameter is stored at a different memory address than the variable used as the argument in the method call. In other words, the actual parameter (the argument in the calling method) and the formal parameter (the parameter in the method header) refer to two separate memory locations, and the called method receives a copy of the sent value. Changes to value parameters never affect the original arguments in calling methods.



A popular advertising campaign declares, "What happens in Vegas, stays in Vegas." The same is true of value parameters within a method—changes to them do not persist outside the method.

Figure 8-1 shows a program that declares a variable named `x`; the figure assumes that `x` is stored at memory address 2000. The value 4 is assigned to `x` and then displayed. Then `x` is passed to a method that accepts a value parameter. The method declares its own local



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



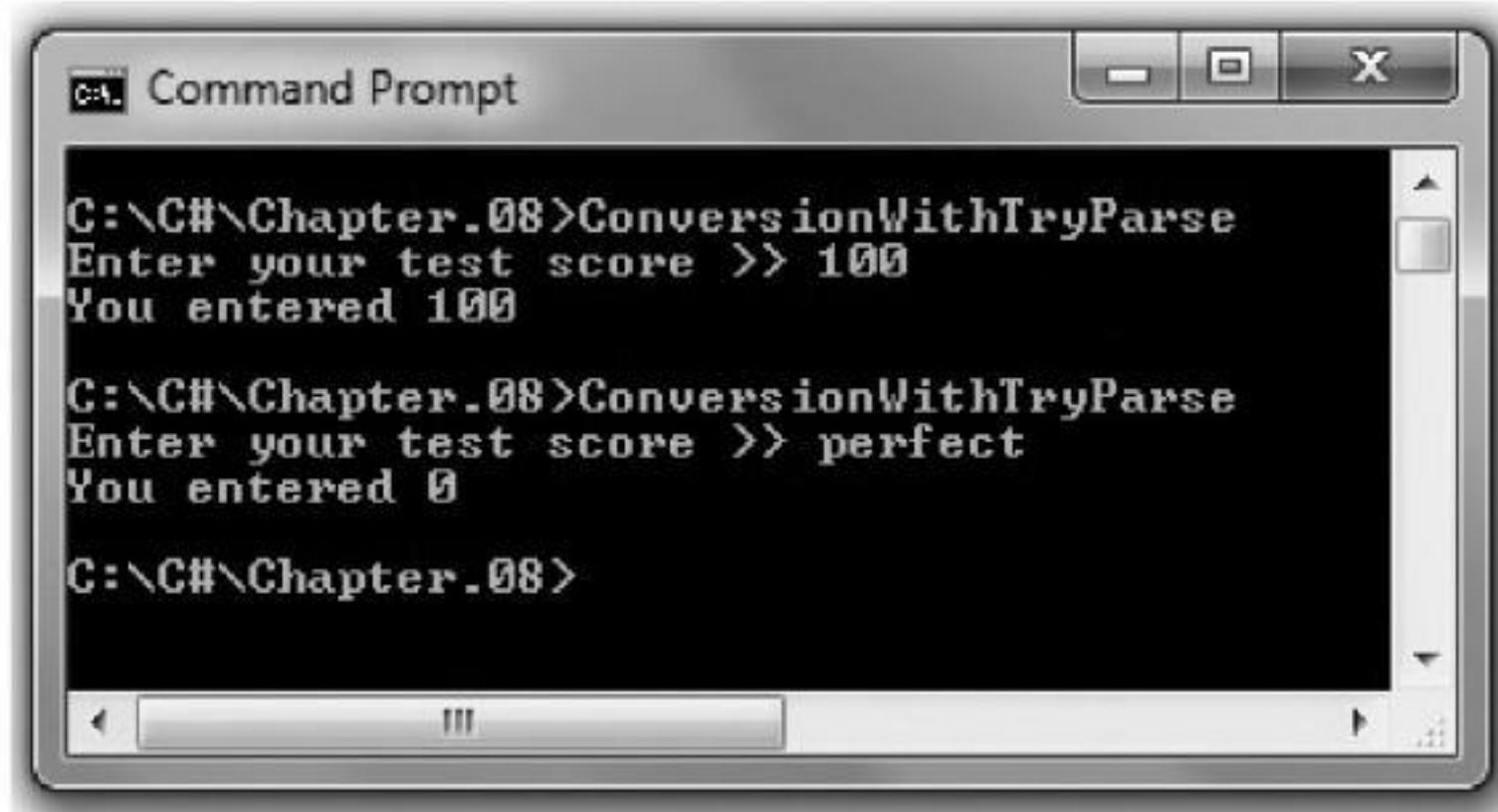
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
using System;
class ConversionWithTryParse
{
    static void Main()
    {
        string entryString;
        int score;
        Console.Write("Enter your test score >> ");
        entryString = Console.ReadLine();
        int.TryParse(entryString, out score);
        Console.WriteLine("You entered {0}", score);
    }
}
```

**Figure 8-9** The ConversionWithTryParse program



**Figure 8-10** Two typical executions of the ConversionWithTryParse program

The TryParse() methods require the receiving variable to be an out parameter because the method can return only one value. The TryParse() methods return a Boolean value that indicates whether the conversion was successful. Suppose that you do not want a score to be assigned 0 if the conversion fails, because 0 is a legitimate score. Instead, you want to assign a -1 if the conversion fails. In that case you can use a statement similar to the following:

```
if(!int.TryParse(entryString, out score))
    score = -1;
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

(continued)

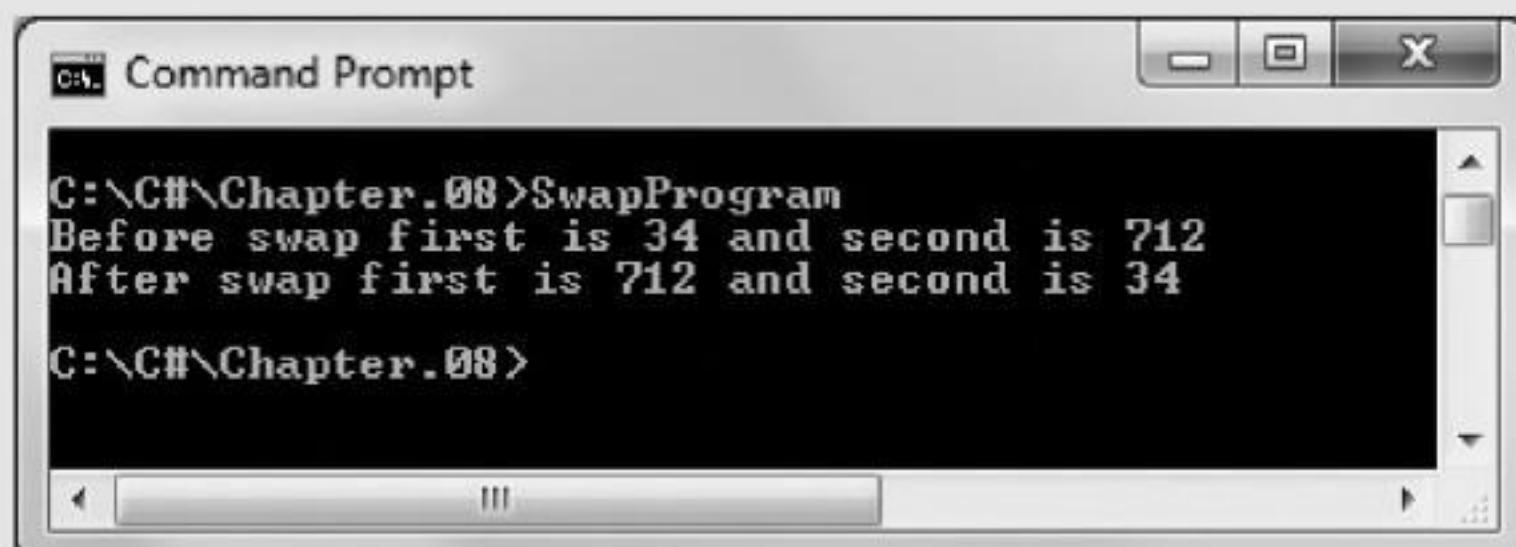
those values in Main(), you can use reference parameters. After the method call, display the two values again. Add the closing curly brace for the Main() method.

```
int first = 34, second = 712;
Console.WriteLine("Before swap first is {0}", first);
Console.WriteLine(" and second is {0}", second);
Swap(ref first, ref second);
Console.WriteLine("After swap first is {0}", first);
Console.WriteLine(" and second is {0}", second);
}
```

3. Create the Swap() method as shown. You can swap two values by storing the first value in a temporary variable, then assigning the second value to the first variable. At this point, both variables hold the value originally held by the second variable. When you assign the temporary variable's value to the second variable, the two values are reversed.

```
private static void Swap(ref int one, ref int two)
{
    int temp;
    temp = one;
    one = two;
    two = temp;
}
```

4. Add the closing curly brace for the class. Save the program, and then compile and execute it. Figure 8-13 shows the output.



**Figure 8-13** Output of the SwapProgram program



You might want to use a module like Swap() as part of a larger program in which you verify, for example, that a higher value is displayed before a lower one; you would include the call to Swap() as part of a decision whose body executes only when a first value is less than a second one.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



In this book you have seen the `Console.WriteLine()` method used with a string parameter, a numeric parameter, and with no parameter. You have also seen it used with several parameters when you use a format string along with several variables. Therefore, you know `Console.WriteLine()` is an overloaded method.

## Understanding Overload Resolution

When a method call could execute multiple overloaded method versions, C# determines which method to execute using a process called **overload resolution**. For example, suppose that you create a method with the following declaration:

```
private static void MyMethod(double d)
```

You can call this method using a `double` argument, as in the following:

```
MyMethod(2.5);
```

You can also call this method using an `int` argument, as in the following:

```
MyMethod(4);
```

The call that uses the `int` argument works because an `int` can automatically be promoted to a `double`. In Chapter 2 you learned that when an `int` is promoted to a `double`, the process is called an *implicit conversion* or *implicit cast*.

Suppose that you create overloaded methods with the following declarations:

```
private static void MyMethod(double d)  
private static void MyMethod(int i)
```

If you then call `MyMethod()` using an integer argument, both methods are **applicable methods**. That means either method on its own could accept a call that uses an `int`. However, if both methods exist in the same class (making them overloaded), the second version will execute because it is a better match for the method call. The rules that determine which method version to call are known as **betterness rules**.

Betterness rules are similar to the implicit conversion rules you learned in Chapter 2. For example, although an `int` could be accepted by a method that accepts an `int`, a `float`, or a `double`, an `int` is the best match. If no method version with an `int` parameter exists, then a `float` is a better match than a `double`. Table 8-1 shows the betterness rules for several data types.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

If you assign a default value to any variable in a method's parameter list, then all parameters to the right of that parameter must also have default values. Table 8-2 shows some examples of valid and invalid method declarations.

Method Declaration	Explanation
<code>private static void M1(int a, int b, int c, int d = 10)</code>	Valid. The first three parameters are mandatory and the last one is optional.
<code>private static void M2(int a, int b = 3, int c)</code>	Invalid. Because <code>b</code> has a default value, <code>c</code> must also have one.
<code>private static void M3(int a = 3, int b = 4, int c = 5)</code>	Valid. All parameters are optional.
<code>private static void M4(int a, int b, int c)</code>	Valid. All parameters are mandatory.
<code>private static void M5(int a = 4, int b, int c = 8)</code>	Invalid. Because <code>a</code> has a default value, both <code>b</code> and <code>c</code> must have default values.

**Table 8-2** Examples of valid and invalid optional parameter method declarations

When you call a method that contains default parameters, you can always choose to provide a value for every parameter. However, if you omit an argument when you call a method that has default parameters, then you must do one of the following:

- You must leave out all unnamed arguments to the right of the last argument you use.
- You can name arguments.

## Leaving Out Unnamed Arguments

When calling a method with optional parameters (and using unnamed arguments), you must leave out any arguments to the right of the last one used. In other words, once an argument is left out, you must leave out all the arguments that would otherwise follow.

For example, assume you have declared a method as follows:

```
private static void Method1(int a, char b, int c = 22, double d = 33.2)
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



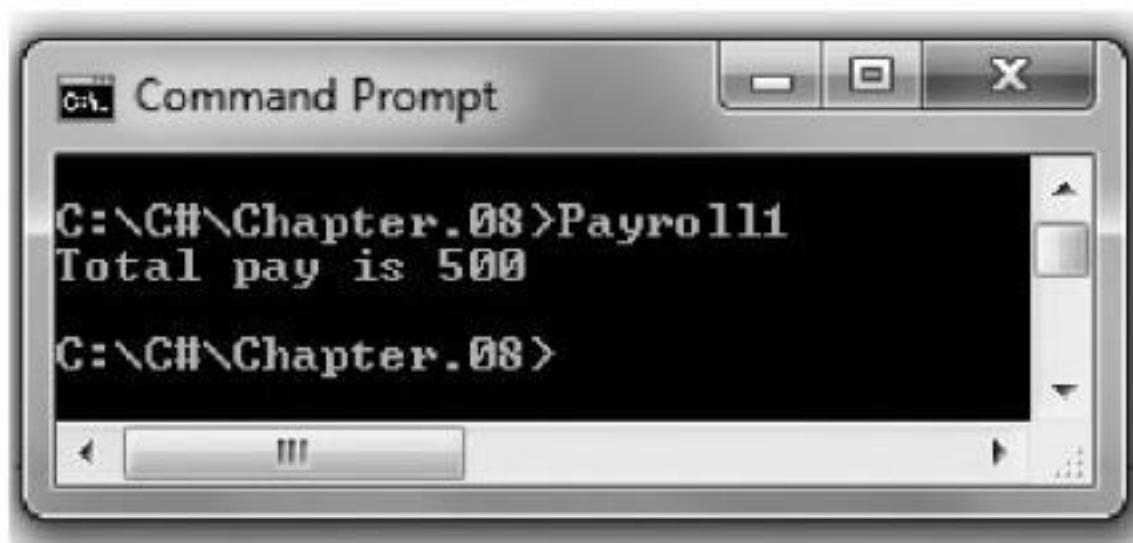
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

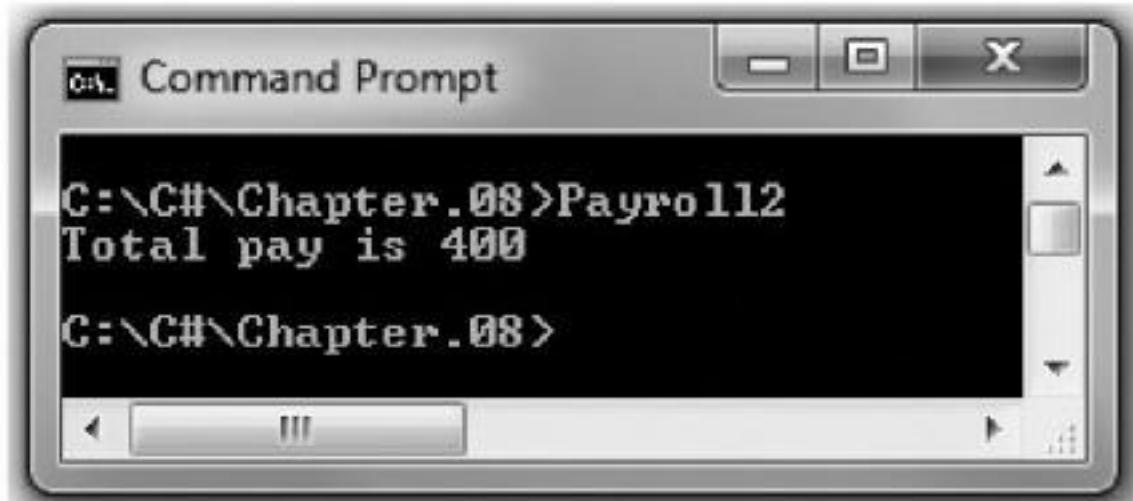


**Figure 8-28** Execution of the Main() method in Figure 8-27

Figure 8-29 shows a Main() method that calls ComputeTotalPay() using named arguments. The only difference from the program in Figure 8-27 is the shaded method call that uses named arguments. In this case, the bonus value sent to ComputeTotalPay() is 0, and then ComputeGross() is called to alter the bonus. However, the bonus alteration is too late and the program output is incorrect, as shown in Figure 8-30.

```
static void Main()
{
    double hours = 40;
    double rate = 10.00;
    double bonus = 0;
    double totalPay;
    totalPay = ComputeTotalPay(bonus: bonus,
        gross: ComputeGross(hours, rate, out bonus));
    Console.WriteLine("Total pay is {0}", totalPay);
}
```

**Figure 8-29** A Main() method that calls ComputeTotalPay() using named arguments



**Figure 8-30** Execution of the Main() method in Figure 8-29

In Chapter 4, you learned that when you combine operands in a Boolean expression using `&&` or `||`, you risk side effects because of short-circuit evaluation. The situation with named arguments is similar. When a named argument is an expression, there can be unintended consequences.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

11. Methods are ambiguous when they \_\_\_\_\_.  
a. are overloaded  
b. are written in a confusing manner  
c. are indistinguishable to the compiler  
d. have the same parameter type as their return type

12. Which of the following pairs of method declarations represent correctly overloaded methods?  
a. `private static void MethodA(int a)`  
`private static void MethodA(int b, double c)`  
b. `private static void MethodB(double d)`  
`private static void MethodB()`  
c. `private static double MethodC(int e)`  
`private static double MethodD(int f)`  
d. Two of these are correctly overloaded methods.

13. Which of the following pairs of method declarations represent correctly overloaded methods?  
a. `private static void Method(int a)`  
`private static void Method(int b)`  
b. `private static void Method(double d)`  
`private static int Method()`  
c. `private static double Method(int e)`  
`private static int Method(int f)`  
d. Two of these are correctly overloaded methods.

14. The process of determining which overloaded version of a method to execute is overload \_\_\_\_\_.  
a. confusion  
b. infusion  
c. revolution  
d. resolution

15. When one of a method's parameters is optional, it means that \_\_\_\_\_.  
a. no arguments are required in a call to the method  
b. a default value will be assigned to the parameter if no argument is sent for it  
c. a default value will override any argument value sent to it  
d. you are not required to use the parameter within the method body



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- b. Create a GUI version of the War card game described in Exercise 11a and name it **WarCardGameGUI**. Let the user click a button to deal the cards, then make that button invisible and expose a Play button. Each time the user clicks Play, a pair of cards is revealed. To keep the `Frame` size reasonable, you might want to erase the output label's contents every four hands or so. The right side of Figure 8-31 shows a typical game in progress.



## Debugging Exercises

1. Each of the following files in the Chapter.08 folder of your downloadable student files has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with *Fixed*. For example, `DebugEight1.cs` will become `FixedDebugEight1.cs`.
  - a. `DebugEight1.cs`
  - b. `DebugEight2.cs`
  - c. `DebugEight3.cs`
  - d. `DebugEight4.cs`



## Case Problems

1. In Chapter 7, you modified the **GreenvilleRevenue** program to include a number of methods. Now modify every data entry statement to use a `TryParse()` method to ensure that each piece of data is the correct type. Any invalid user entries should generate an appropriate message, and the user should be required to reenter the data.
2. In Chapter 7, you modified the **MarshallsRevenue** program to include a number of methods. Now modify every data entry statement to use a `TryParse()` method to ensure that each piece of data is the correct type. Any invalid user entries should generate an appropriate message, and the user should be required to reenter the data.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

## Creating a Class from Which Objects Can Be Instantiated

372

When you create a class, you must assign a name to it and determine what data and methods will be part of the class. For example, suppose that you decide to create a class named `Employee`. One instance variable of `Employee` might be an employee number, and one necessary method might display a welcome message to new employees. To begin, you create a **class header** or **class definition** that starts the class. It contains three parts:

1. An optional access modifier
2. The keyword `class`
3. Any legal identifier you choose for the name of your class; because each class represents a type of object, class names are usually singular nouns.

You will learn about other optional components that can be added to a class definition as you continue to study C#.

For example, one usable header for an `Employee` class is `internal class Employee`. The keyword `internal` is an example of a **class access modifier**. You can declare a class using any of the modifiers in Table 9-1.

---

Class Access Modifier	Description
<code>public</code>	Access to the class is not limited.
<code>protected</code>	Access to the class is limited to the class and to any classes derived from the class. (You will learn about deriving classes in the chapter <i>Introduction to Inheritance</i> .)
<code>internal</code>	Access is limited to the assembly to which the class belongs. (An <b>assembly</b> is a group of code modules compiled together to create an executable program. The .exe files you create after compiling a C# program are assemblies.)
<code>private</code>	Access is limited to another class to which the class belongs. In other words, a class should be <code>private</code> if it is contained within another class, and the containing class is the only one that should have access to it.

---

**Table 9-1** Class access modifiers



When you declare a class using a namespace, you only can declare it to be `public` or `internal`.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

A property declaration resembles a variable declaration; it contains an access modifier, a data type, and an identifier. It also resembles a method in that it is followed by curly braces that contain statements. By convention, a property identifier is the same as the field it manipulates, except the first letter is capitalized. The field that supports a property is its **backing field**. Be careful with capitalization of properties. For example, within a **get** accessor for **IdNumber**, if you return the property (**IdNumber**) instead of the backing field (**idNumber**), you initiate an infinite loop—the property continuously accesses itself.

The **IdNumber** property in Figure 9-8 contains both **get** and **set** accessors defined between curly braces; a property declaration can contain a **get** accessor, a **set** accessor, or both. When a property has a **set** accessor, programmers say the property can be “written to.” When it has a **get** accessor, programmers say the property can be “read from.” When a property has only a **get** accessor (and not a **set** accessor), it is a **read-only property**. In C#, the **get** and **set** accessors often are called the **getter** and the **setter**, respectively.



In the **WelcomeMessage()** method in Figure 9-8, the **IdNumber** property is displayed. Alternatively, this method could continue to use the **idNumber** field (as in Figure 9-3) because the method is a member of the same class as the field. Programmers are divided on whether a method of a class should use a field or a property to access its own methods. One popular position is that if **get** and **set** accessors are well-designed, they should be used everywhere, even within the class. Sometimes, you want a field to be read-only, so you do not create a **set** accessor. In such a case, you can use the field (with the lowercase initial by convention) within class methods.



Throughout this book you have seen keywords displayed in boldface in the program figures. The words **get**, **set**, and **value** are not C# keywords—for example, you could declare a variable named **get** within a C# program. However, within a property, **get**, **set**, and **value** have special meanings and are not allowed to be declared as identifiers there. In the Visual Studio Integrated Development Environment, these three words appear in blue within properties, but in black elsewhere. Identifiers that act like keywords in specific circumstances are **contextual keywords**. C# has six contextual keywords: **get**, **set**, **value**, **partial**, **where**, and **yield**.

Each accessor in a property looks like a method, except no parentheses are included in the identifier. A **set** accessor acts like a method that accepts a parameter and assigns it to a variable. However, it is not a method and you do not use parentheses with it. A **get** accessor returns the value of the field associated with the property, but you do not code a **return** type; the **return** type of a **get** accessor is implicitly the type of the property in which it is contained.

When you use **set** and **get** accessors in a method, you do not use the words **set** or **get**. Instead, to set a value, you use the assignment operator (=), and to get a value, you simply use the property name. For example, if you declare an **Employee** named **myChef**, you can assign an **IdNumber** as simply as you would a variable, as in the following:

```
Employee myChef = new Employee();
myChef.IdNumber = 2345;
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

*(continued)*

6. At the top of the file, begin a program that creates two `Student` objects, assigns some values, and displays the `Students`.

```
using System;
class CreateStudents
{
```

7. Add a `Main()` method that declares two `Students`. Assign field values, including one “illegal” value—a grade point average that is too high.

```
static void Main()
{
    Student first = new Student();
    Student second = new Student();
    first.IdNumber = 123;
    first.LastName = "Anderson";
    first.GradePointAverage = 3.5;
    second.IdNumber = 789;
    second.LastName = "Daniels";
    second.GradePointAverage = 4.1;
```

8. Instead of creating similar `WriteLine()` statements to display the two `Students`, call a method with each `Student`. You will create the method to accept a `Student` argument in the next step. Add a closing curly brace for the `Main()` method.

```
Display(first);
Display(second);
}
```

9. Write the `Display()` method so that the passed-in `Student`’s `IdNumber`, `LastName`, and `GradePointAverage` are displayed and aligned. Add a closing curly brace for the class.

```
static void Display(Student stu)
{
    Console.WriteLine("{0,5} {1,-10}{2,6}",
                      stu.IdNumber, stu.LastName,
                      stu.GradePointAverage.ToString("F1"));
}
```



Recall that field contents are left-aligned when you use a minus sign before the field size. Also recall that the “F1” argument to the `ToString()` method causes the value to be displayed to one decimal place.

*(continues)*



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



**Figure 9-16** Output of the TestCarpet program

The Carpet class contains one private method named `CalcArea()`. As you examine the code in the `TestCarpet` class in Figure 9-15, notice that `Width` and `Length` are set using an assignment operator, but `Area` is not. The `TestCarpet` class can make assignments to `Width` and `Length` because these properties are `public`. However, you would not want a client program to assign a value to `Area` because the assigned value might not agree with the `Width` and `Length` values. Therefore, the `Area` property is a read-only property—it does not contain a `set` accessor, and no assignments by clients are allowed. Instead, whenever the `Width` or `Length` properties are set, the `private CalcArea()` method is called from the accessor. The `CalcArea()` method is defined as `private` because there is no reason for a client class like `TestCarpet` to call `CalcArea()`. Programmers probably create `private` methods more frequently than they create `public` data fields. Some programmers feel that the best style is to use `public` methods that are nothing but a list of method calls with descriptive names. Then, the methods that actually do the work are all `private`.

## TWO TRUTHS & A LIE

### More About public and private Access Modifiers

1. Good object-oriented techniques require that data should usually be hidden and access to it should be controlled by well-designed accessors.
2. Although `private` fields, methods, and accessors are the norm, occasionally you need to create `public` versions of them.
3. When you define a named constant within a class, it is always `static`; that is, the field belongs to the entire class, not to any particular instance of the class.

The false statement is #2. Although `private` fields and `public` methods and accessors are the norm, occasionally you need to create `public` fields or `private` methods.

## Understanding the `this` Reference

When you create a class, only one copy of the class code is stored in computer memory. However, you might eventually create thousands of objects from the class. When you create each object, you provide storage for each of the object's instance variables. For example,



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

class Book
{
    private double price;
    private double tax;
    private double price;
    public void SetPriceAndTax(double price)
    {
        const double TAX_RATE = 0.07;
        this.price = price;
        tax = price * TAX_RATE;
    }
}

```

The field price is **this.price** within the method.

The parameter price is just price within the method.

**Figure 9-21** Book class that must explicitly use the **this** reference



Watch the video *Understanding the this Reference*.

## TWO TRUTHS & A LIE

### Understanding the **this** Reference

1. An implicit, or invisible, **this** reference is passed to every instance method and property accessor in a class; instance methods and properties are nonstatic.
2. You can explicitly refer to the **this** reference within an instance method or property.
3. Although the **this** reference exists in every instance method, you can never refer to it within a method.

The false statement is #3. Sometimes, you may want to include the **this** reference within a method for clarity, so the reader has no doubt when you are referring to a class instance variable.

## Understanding Constructors

When you create a class such as **Employee** and instantiate an object with a statement such as **Employee aWorker = new Employee();**, you are actually calling a method named **Employee()** that is provided by C#. A **constructor** is a method that instantiates (creates an instance of) an object. If you do not write a constructor for a class, then each class you create is



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

(continued)

403

```
public Student(int id, string name, double gpa)
{
    IdNumber = id;
    LastName = name;
    GradePointAverage = gpa;
}
```

2. Add a second parameterless constructor. It calls the first constructor, passing 0 for the ID number, "XXX" for the name, and 0.0 for the grade point average. Its body is empty.

```
public Student() : this(0, "XXX", 0.0)
{
}
```

3. Change the name of the `CreateStudents` class to `CreateStudents2`.
4. After the existing declarations of the `Student` objects, add two more declarations. With one, use three arguments, but with the other, do not use any.

```
Student third = new Student(456, "Marco", 2.4);
Student fourth = new Student();
```

5. At the end of the `Main()` method, just after the two existing calls to the `Display()` method, add two more calls using the new objects:

```
Display(third);
Display(fourth);
```

6. Save the file, and then compile and execute it. The output looks like Figure 9-28. All four objects are displayed. The first two have had values assigned to them after declaration, but the third and fourth ones obtained their values from their constructors.

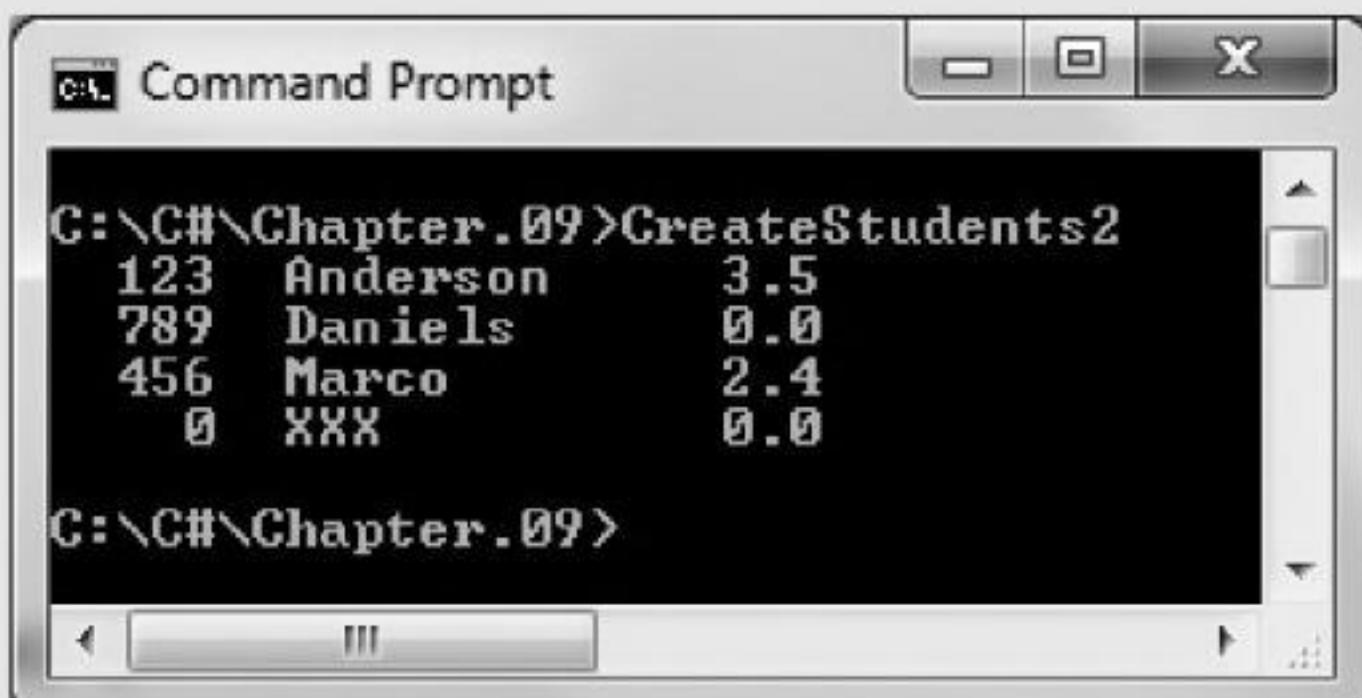


Figure 9-28 Output of `CreateStudents2` program



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Just as it is convenient to use a + between both integers and doubles to add them, it also can be convenient to use a + between objects, such as Employees or Books, to add them. To be able to use arithmetic symbols with your own objects, you must overload the symbols.

408

C# operators are classified as unary or binary, depending on whether they take one or two arguments, respectively. The rules for overloading are shown in the following list.

- The overloadable unary operators are:  
`+ - ! ~ ++ -- true false`
- The overloadable binary operators are:  
`+ - * / % & | ^ == != > < >= <=`
- You cannot overload the following operators:  
`= && || ?? ?: checked unchecked new typeof as is`
- You cannot overload an operator for a built-in data type. For example, you cannot change the meaning of + between two ints.
- When a binary operator is overloaded and it has a corresponding assignment operator, it is also overloaded. For example, if you overload +, then += is automatically overloaded too.
- Some operators must be overloaded in pairs. For example, when you overload ==, you also must overload !=, and when you overload >, you also must overload <.



When you overload ==, you also receive warnings about methods in the `Object` class. You will learn about this class in the chapter *Introduction to Inheritance*; you should not attempt to overload == until you have studied that chapter.

You have used many, but not all, of the operators listed above. If you want to include an overloaded operator in a class, you must decide what the operator will mean in your class. When you do, you write statements in a method to carry out your meaning. The method has a return type and arguments just like other methods, but its identifier is required to be named `operator` followed by the operator being overloaded—for example, `operator+()` or `operator*()`.

For example, suppose that you create a Book class in which each object has a title, number of pages, and a price. Further assume that, as a publisher, you have decided to “add” Books together. That is, you want to take two existing Books and combine them into one. Assume that you want the new book to have the following characteristics:

- The new title is a combination of the old titles, joined by the word “and.”
- The number of pages in the new book is equal to the sum of the pages in the original Books.
- Instead of charging twice as much for a new Book, you have decided to charge the price of the more expensive of the two original Books, plus \$10.

A different publisher might have decided that “adding Books” means something different—for example, an added Book might have a fixed new price of \$29.99. The statements you write in



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

overload an operator, you should write statements that intuitively have the same meaning as the common use of the operator.

- You can declare arrays of references to objects. After doing so, you must call a constructor to instantiate each object. To use a method that belongs to an object that is part of an array, you insert the appropriate subscript notation after the array name and before the dot-method. The `IComparable` interface contains the definition for the `CompareTo()` method; you override this method to tell the compiler how to compare objects.
- A destructor contains the actions you require when an instance of a class is destroyed. If you do not explicitly create a destructor for a class, C# automatically provides one. To explicitly declare a destructor, you use an identifier that consists of a tilde (~) followed by the class name. You cannot provide any parameters to a destructor; a class can have one destructor at most.
- GUI objects such as `Forms`, `Buttons`, and `Labels` are typical C# objects that encapsulate properties and methods. When you start a Windows Forms application and drag a component onto a `Form`, statements are automatically created by the IDE to instantiate the object and assign values to some of its properties.

## Key Terms

To **instantiate** an object is to create it.

An **instantiation** of a class is a created object.

An **instance** of a class is one object, or one instantiation.

The **instance variables** of a class are the data components that exist separately for each instantiation.

**Fields** are instance variables within a class.

An object's **state** is the set of contents of its fields.

**Instance methods** are methods that are used with object instantiations.

A **class client** or **class user** is a program or class that instantiates objects of another prewritten class.

A **class header** or **class definition** describes a class; it contains an optional access modifier, the keyword `class`, and any legal identifier for the name of the class.

A **class access modifier** describes access to a class.

The **public** class access modifier means access to the class is not limited.

The **protected** class access modifier means access to the class is limited to the class and to any classes derived from the class.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

b. Harold has realized that his method for computing the fee for combined jobs is not fair. For example, consider the following:

- His fee for painting a house is \$100 per hour. If a job takes 10 hours, he earns \$1000.
- His fee for dog walking is \$10 per hour. If a job takes 1 hour, he earns \$10.
- If he combines the two jobs and works a total of 11 hours, he earns only the average rate of \$55 per hour, or \$605.

Devise an improved, weighted method for calculating Harold's fees for combined jobs and include it in the overloaded `operator+()` method. Write a program named **DemoJobs2** that demonstrates all the methods in the class work correctly.

11. a. Write a **FractionDemo** program that instantiates several **Fraction** objects and demonstrates that their methods work correctly. Create a **Fraction** class with fields that hold a whole number, a numerator, and a denominator. In addition:

- Create properties for each field. The `set` accessor for the denominator should not allow a 0 value; the value defaults to 1.
- Add three constructors. One takes three parameters for a whole number, numerator, and denominator. Another accepts two parameters for the numerator and denominator; when this constructor is used, the whole number value is 0. The last constructor is parameterless; it sets the whole number and numerator to 0 and the denominator to 1. (After construction, **Fractions** do not have to be reduced to proper form. For example, even though  $\frac{3}{9}$  could be reduced to  $\frac{1}{3}$ , your constructors do not have to perform this task.)
- Add a `Reduce()` method that reduces a **Fraction** if it is in improper form. For example,  $\frac{2}{4}$  should be reduced to  $\frac{1}{2}$ .
- Add an `operator+()` method that adds two **Fractions**. To add two fractions, first eliminate any whole number part of the value. For example,  $2\frac{1}{4}$  becomes  $\frac{9}{4}$  and  $1\frac{3}{5}$  becomes  $\frac{8}{5}$ . Find a common denominator and convert the fractions to it. For example, when adding  $\frac{9}{4}$  and  $\frac{8}{5}$ , you can convert them to  $\frac{45}{20}$  and  $\frac{32}{20}$ . Then you can add the numerators, giving  $\frac{77}{20}$ . Finally, call the `Reduce()` method to reduce the result, restoring any whole number value so the fractional part of the number is less than 1. For example,  $\frac{77}{20}$  becomes  $3\frac{17}{20}$ .
- Include a function that returns a `string` that contains a **Fraction** in the usual display format—the whole number, a space, the numerator, a slash (/), and a denominator. When the whole number is 0, just the **Fraction** part of the value should be displayed (for example,  $1/2$  instead of  $0\frac{1}{2}$ ). If the numerator is 0, just the whole number should be displayed (for example,  $2$  instead of  $2\frac{0}{3}$ ).



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
class Employee
{
    private int empNum;
    private double empSal;
    public int EmpNum
    {
        get
        {
            return empNum;
        }
        set
        {
            empNum = value;
        }
    }
    public double EmpSal
    {
        get
        {
            return empSal;
        }
        set
        {
            empSal = value;
        }
    }
    public string GetGreeting()
    {
        string greeting = "Hello. I am employee #" + EmpNum;
        return greeting;
    }
}
```

**Figure 10-1** The Employee class

After you create the `Employee` class, you can create specific `Employee` objects, as in the following:

```
Employee receptionist = new Employee();
Employee deliveryPerson = new Employee();
```

These `Employee` objects can eventually possess different numbers and salaries, but because they are `Employee` objects, you know that each possesses *some* number and salary.

Suppose that you hire a new type of `Employee` who earns a commission as well as a salary. You can create a class with a name such as `CommissionEmployee`, and provide this class with three fields (`empNum`, `empSal`, and `commissionRate`), three properties (with accessors to get and set each of the three fields), and a greeting method. However, this work would duplicate much of the work that you already have done for the `Employee` class. The wise and efficient alternative is to create the class `CommissionEmployee` so it inherits all the attributes and methods of



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

(continued)

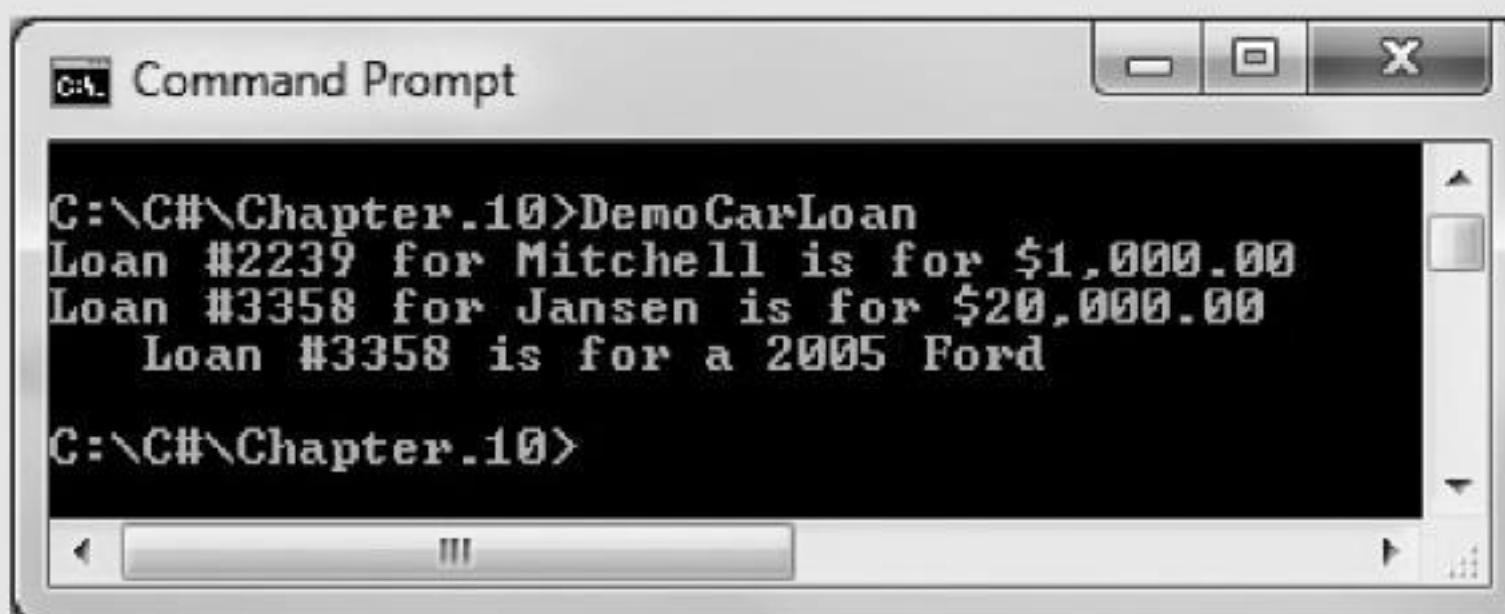
3. After the three property assignments for the `Loan` object, insert five assignment statements for the `CarLoan` object.

```
aCarLoan.LoanNumber = 3358;  
aCarLoan.LastName = "Jansen";  
aCarLoan.LoanAmount = 20000.00;  
aCarLoan.Make = "Ford";  
aCarLoan.Year = 2005;
```

4. Following the `WriteLine()` statement that displays the `Loan` object data, insert two `WriteLine()` statements that display the `CarLoan` object's data.

```
Console.WriteLine("Loan #{0} for {1} is for {2}",  
    aCarLoan.LoanNumber, aCarLoan.LastName,  
    aCarLoan.LoanAmount.ToString("C2"));  
Console.WriteLine("    Loan #{0} is for a {1} {2}",  
    aCarLoan.LoanNumber, aCarLoan.Year,  
    aCarLoan.Make);
```

5. Save the program, and then compile and execute it. The output looks like Figure 10-7. The `CarLoan` object correctly uses its own fields and properties as well as those of the parent `Loan` class.



**Figure 10-7** Output of the `DemoCarLoan` program

## Using the `protected` Access Specifier

The `Employee` class in Figure 10-1 is a typical C# class in that its data fields are `private` and its properties and methods are `public`. In the chapter *Using Classes and Objects*, you learned that this scheme provides for information hiding—protecting your `private` data from alteration by methods outside the data's own class. When a program is a client of the `Employee` class (that is, it instantiates an `Employee` object), the client cannot alter the data in any `private` field directly. For example, when you write a `Main()` method that creates an `Employee` named `clerk`, you cannot change the `Employee`'s `empNum` or `empSal` directly using a statement such as `clerk.empNum = 2222;`. Instead, you must use the `public` `EmpNum` property to set the `empNum` field of the `clerk` object.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

## TWO TRUTHS & A LIE

### Using the **protected** Access Specifier

1. A child class does not possess the private members of its parent.
2. A child class cannot use the private members of its parent.
3. A child class can use the protected members of its parent, but outside classes cannot.

The false statement is #1. A child class possesses the private members of its

parent, but cannot use them directly.

## Overriding Base Class Members

When you create a derived class by extending an existing class, the new derived class contains data and methods that were defined in the original base class. Sometimes, the superclass fields, properties, and methods are not entirely appropriate for the subclass objects. Using the same method or property name to indicate different implementations is called polymorphism. The word *polymorphism* means “many forms”—in programming, it means that many forms of action take place, even though you use the same method name. The specific method executed depends on the object.



You first learned the term *polymorphism* in Chapter 1.

Everyday cases provide many examples of polymorphism:

- Although both are musical instruments and have a `Play()` method, a guitar is played differently than a drum.
- Although both are vehicles and have an `Operate()` method, a bicycle is operated differently than a truck.
- Although both are schools and have a `SatisfyGraduationRequirements()` method, a preschool's requirements are different from those of a college.

You understand each of these methods based on the context in which it is used. In a similar way, C# understands your use of the same method name based on the type of object associated with it.

For example, suppose that you have created a `Student` class as shown in Figure 10-11. Students have names, credits for which they are enrolled, and tuition amounts. You can set a `Student`'s name and credits by using the `set` accessors in the `Name` and `Credits` properties,



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



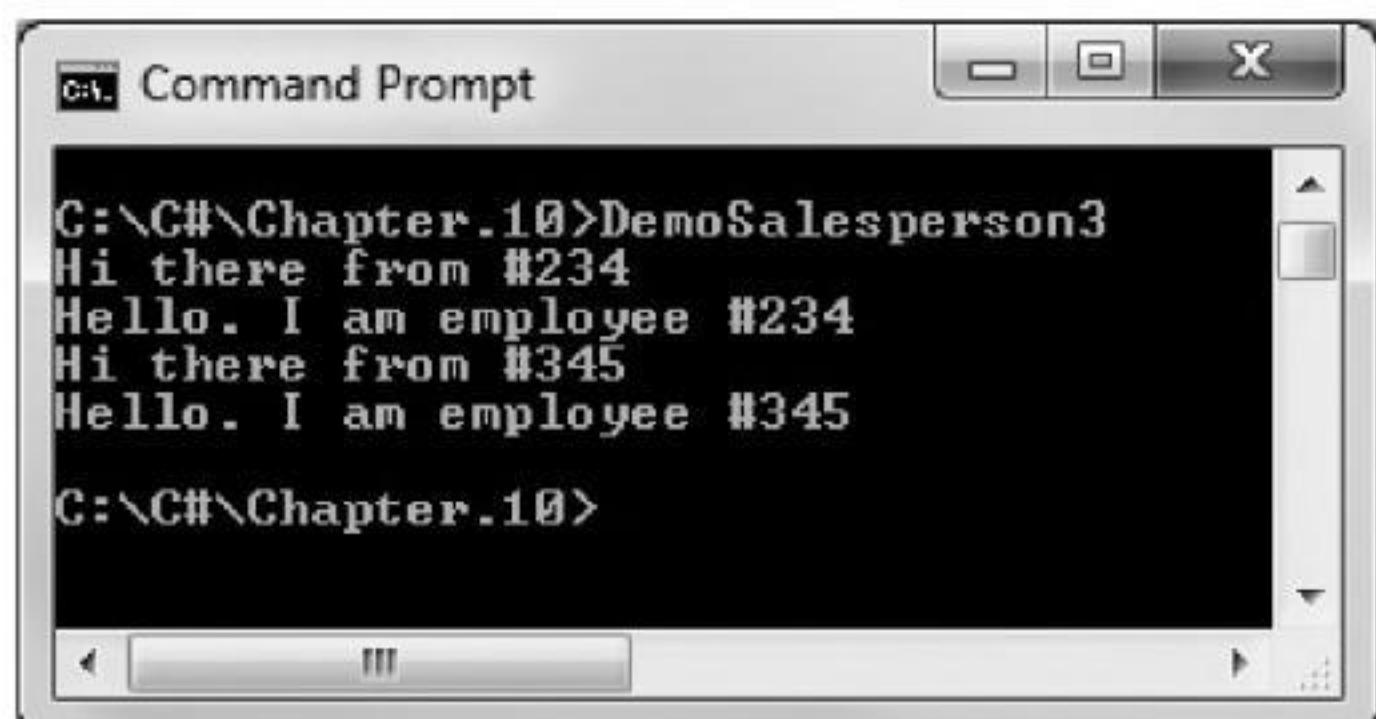
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

the object is treated as though it had only the characteristics defined in the base class and not those added in the child class definition.

For example, when a `CommissionEmployee` class inherits from `Employee`, an object of either type can be passed to a method that accepts an `Employee` parameter. In Figure 10-19, an `Employee` is passed to `DisplayGreeting()` in the first shaded statement, and a `CommissionEmployee` is passed in the second shaded statement. Each is referred to as `emp` within the method, and each is used correctly, as shown in Figure 10-20.

```
using System;
class DemoSalesperson3
{
    static void Main()
    {
        Employee clerk = new Employee();
        CommissionEmployee salesperson = new CommissionEmployee();
        clerk.EmpNum = 234;
        salesperson.EmpNum = 345;
        DisplayGreeting(clerk);
        DisplayGreeting(salesperson);
    }
    public static void DisplayGreeting(Employee emp)
    {
        Console.WriteLine("Hi there from #" + emp.EmpNum);
        Console.WriteLine(emp.GetGreeting());
    }
}
```

**Figure 10-19** The DemoSalesperson3 program



**Figure 10-20** Output of the DemoSalesperson3 program



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Abstract classes usually contain abstract methods, although methods are not required. An **abstract method** has no method statements; any class derived from a class that contains an abstract method must override the abstract method by providing a body (an implementation) for it. (Alternatively, the derived class can declare the method to be abstract; in that case, the derived class's children must implement the method.) You can create an **abstract class** with no **abstract** methods, but you cannot create an **abstract** method outside of an **abstract** class.



A method that is declared **virtual** is not required to be overridden in a child class, but a method declared **abstract** must be overridden.

When you create an abstract method, you provide the keyword **abstract** and the intended method type, name, and parameters, but you do not provide statements within the method; you do not even supply curly braces. When you create a derived class that inherits an abstract method from a parent, you must use the keyword **override** in the method header and provide the actions, or implementation, for the inherited method within the derived class. In other words, you are required to code a derived class method to override any empty base class methods that are inherited.

For example, suppose that you want to create classes to represent different animals. You can create a generic, abstract class named **Animal** so you can provide generic data fields, such as the animal's name, only once. An **Animal** is generic, but each specific **Animal**, such as **Dog** or **Cat**, makes a unique sound. If you code an abstract **Speak()** method in the abstract **Animal** class, then you require all future **Animal** derived classes to override the **Speak()** method and provide an implementation that is specific to the derived class. Figure 10-31 shows an abstract **Animal** class that contains a data field for the name, a constructor that assigns a name, a **Name** property, and an abstract **Speak()** method.

```
abstract class Animal
{
    private string name;
    public Animal(string name)
    {
        this.name = name;
    }

    public string Name
    {
        get
        {
            return name;
        }
    }

    public abstract string Speak();
}
```

Figure 10-31 The Animal class



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
class Employee : IWorkable
{
    public Employee(string name)
    {
        Name = name;
    }
    public string Name {get; set;}
    public string Work()
    {
        return "I do my job";
    }
}
abstract class Animal : IWorkable
{
    public Animal(string name)
    {
        Name = name;
    }
    public string Name {get; set;}

    public abstract string Work();
}
class Dog : Animal
{
    public Dog(string name) : base(name)
    {
    }
    public override string Work()
    {
        return "I watch the house";
    }
}
class Cat : Animal
{
    public Cat(string name) : base(name)
    {
    }
    public override string Work()
    {
        return "I catch mice";
    }
}
```

**Figure 10-37** The Employee, Animal, Cat, and Dog classes with the IWorkable interface

When you create a program that instantiates an Employee, a Dog, or a Cat, as in the Demoworking program in Figure 10-38, each object type knows how to “Work()” appropriately. Figure 10-39 shows the output.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

## TWO TRUTHS & A LIE

### Recognizing Inheritance in GUI Applications and Recapping the Benefits of Inheritance

1. Inheritance enables you to create powerful computer programs more easily.
2. Without inheritance, you *could* create every part of a program from scratch, but reusing existing classes and interfaces makes your job easier.
3. Inheritance is frequently inefficient because base class code is seldom reliable when extended to a derived class.

The false statement is #3. Derived class creators save testing time because the base class code already has been tested and probably used in a variety of situations. In other words, the base class code is reliable.

## Chapter Summary

- The classes you create in object-oriented programming languages can inherit data and methods from existing classes. The ability to use inheritance makes programs easier to write, easier to understand, and less prone to errors. A class that is used as a basis for inheritance is called a base class, superclass, or parent class. When you create a class that inherits from a base class, it is called a derived class, extended class, subclass, or child class.
- When you create a class that is an extension or child of another class, you use a single colon between the derived class name and its base class name. The child class inherits all the methods and fields of its parent. Inheritance works only in one direction—a child inherits from a parent, but not the other way around.
- If you could use private data outside of its class, the principle of information hiding would be destroyed. On some occasions, however, you want to access parent class data from a derived class. For those occasions, you declare parent class fields using the keyword **protected**, which provides you with an intermediate level of security between **public** and **private** access.
- When you declare a child class method with the same name and parameter list as a method within its parent class, you override the parent class method and allow your class objects to exhibit polymorphic behavior. You can use the keyword **new** or **override** with the derived class method. When a derived class overrides a parent class member but you want to access the parent class version, you can use the keyword **base**.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

(continued)

- No error message window appears (as in Figure 11-3).
- The error messages displayed are cleaner and “friendlier” than the automatically generated versions in Figure 11-4.
- The program ends normally in each case, with the `answer` value displayed in a user-friendly manner.

517

```
C:\>C#\Chapter.11>ExceptionsOnPurpose2
Enter an integer >> seven
Input string was not in a correct format.
The answer is 0

C:\>C#\Chapter.11>ExceptionsOnPurpose2
Enter an integer >> 7.7
Input string was not in a correct format.
The answer is 0

C:\>C#\Chapter.11>ExceptionsOnPurpose2
Enter an integer >> 7
Attempted to divide by zero.
The answer is 7

C:\>C#\Chapter.11>
```

**Figure 11-8** Error messages generated by successive executions of the `ExceptionsOnPurpose2` program

## Using the Exception Class's `ToString()` Method and Message Property

When the `MilesPerGallon2` program displays the error message (“You attempted to divide by zero!”), you actually cannot confirm from the message that division by zero was the source of the error. In reality, any `Exception` type generated within the `try` block of the program would be caught by the `catch` block in the method because the argument in the `catch` block is a general `Exception`, which is the parent class to all the `Exception` subtypes.

Instead of writing your own message, you can use the `ToString()` method that every `Exception` object inherits from the `Object` class. The `Exception` class overrides `ToString()` to provide a descriptive error message so a user can receive precise information about the nature of any `Exception` that is thrown. For example, Figure 11-9 shows a `MilesPerGallon3` program. The shaded areas show the only changes from the `MilesPerGallon2` program: the name of the class and the message that is displayed when an `Exception` is thrown. In this example, the `ToString()` method is used with the caught `Exception e`. Figure 11-10 shows an execution of the program in which the user enters 0 for `gallonsOfGas`.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



## You Do It

### Catching Multiple Exception Types

In this section, you instigate multiple outcomes by providing multiple catch blocks for code in a try block.

525

1. Open the **ExceptionsOnPurpose2** program if it is not still open. Change the class name to **ExceptionsOnPurpose3**, and immediately save the project as **ExceptionsOnPurpose3**.
2. Replace the existing generic catch block with two catch blocks. (The statement that displays the answer still follows these catch blocks.) The first catches any `FormatException` and displays a short message. The second catches a `DivideByZeroException` and displays a much longer message.

```
catchFormatException e)
{
    Console.WriteLine("You did not enter an integer");
}
catch(DivideByZeroException e)
{
    Console.WriteLine("This is not your fault.");
    Console.WriteLine("You entered the integer correctly.");
    Console.WriteLine("The program divides by zero.");
}
```

3. Save the program and compile it. When you execute the program and enter a value that is not an integer, the first catch block executes. When you enter an integer so that the program can proceed to the statement that divides by 0, the second catch block executes. Figure 11-18 shows two typical executions of the program.

(continues)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



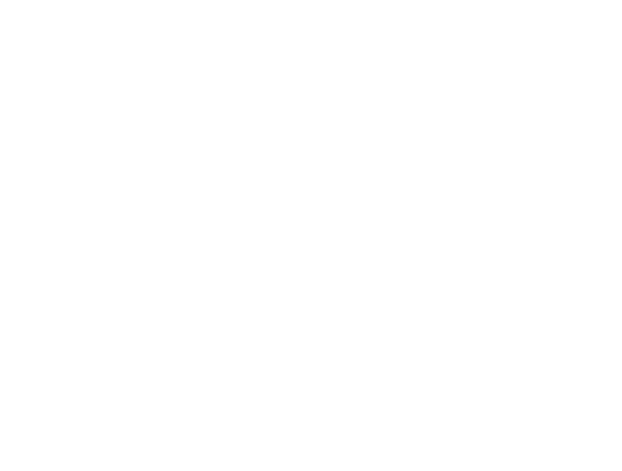
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

words, the GUI objects the user sees on the screen. This class handles user input through the keyboard and pointing devices as well as message routing and security. It defines the bounds of a **Control** by determining its position and size.

Table 12-1 shows the 26 direct descendants of **Control** and some commonly used descendants of those classes. It does not show all the descendants that exist; rather, it shows only the descendants covered previously or in this chapter. For example, the **ButtonBase** class is the parent of **Button**, a class you have used throughout this book. In this chapter, you will use two other **ButtonBase** children—**CheckBox** and **RadioButton**. This chapter cannot cover every **Control** that has been invented; however, after you learn to use some **Controls**, you will find that others work in much the same way. You also can read more about them in the Visual Studio Help documentation.

Class	Commonly used descendants
<code>Microsoft.WindowsCE.Forms.DocumentList</code>	
<code>System.Windows.Forms.AxHost</code>	
<code>System.Windows.Forms.ButtonBase</code>	<code>Button, CheckBox, RadioButton</code>
<code>System.Windows.Forms.DataGrid</code>	
<code>System.Windows.Forms.DataGridView</code>	
<code>System.Windows.Forms.DateTimePicker</code>	
<code>System.Windows.Forms.GroupBox</code>	
<code>System.Windows.Forms.Integration.ElementHost</code>	
<code>System.Windows.Forms.Label</code>	<code>LinkLabel</code>
<code>System.Windows.Forms.ListControl</code>	<code>ListBox, ComboBox, CheckedListBox</code>
<code>System.Windows.Forms.ListView</code>	
<code>System.Windows.Forms.MdiClient</code>	
<code>System.Windows.Forms.MonthCalendar</code>	
<code>System.Windows.Forms.PictureBox</code>	
<code>System.Windows.Forms.PrintPreviewControl</code>	
<code>System.Windows.Forms.ProgressBar</code>	
<code>System.Windows.Forms.ScrollableControl</code>	
<code>System.Windows.Forms.ScrollBar</code>	
<code>System.Windows.Forms.Splitter</code>	
<code>System.Windows.Forms.StatusBar</code>	

**Table 12-1** Classes derived from `System.Windows.Forms.Control` (continues)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

(continued)

3. Next, change the BackColor property of the Bailey's Bed and Breakfast Form. Click the **Form1.cs[Design]** tab and click the **Form**, or click the list box of components at the top of the Properties window and select **BaileysForm**. In the Properties list, click the **BackColor** property to see its list of choices. Choose the **Custom** tab and select **Yellow** in the third row of available colors. Click the **Form**, notice the color change, and then view the code in the **Form1.Designer.cs** file. Locate the statement that changes the **BackColor** of the **Form** to **Yellow**. As you continue to design **Forms**, periodically check the code to confirm your changes and better learn C#.
4. Save the project.
5. If you want to take a break at this point, close Visual Studio. You return to this project in the "You Do It" section at the end of this chapter.

## Using CheckBox and RadioButton Objects

**CheckBox** and **RadioButton** widgets allow users to select an option by clicking a small square or circle beside a label. Like **Button**, both classes descend from **ButtonBase**. When a **Form** contains multiple **Checkboxes**, any number of them can be checked or unchecked at the same time. **RadioButtons** differ in that only one **RadioButton** in a group can be selected at a time—selecting any **RadioButton** automatically deselects the others.

Table 12-6 contains commonly used **CheckBox** and **RadioButton** properties and the default event for which a method shell is generated when you double-click a **CheckBox** or **RadioButton** in the IDE.

Property or Method	Description
<b>Checked</b>	Indicates whether the <b>CheckBox</b> or <b>RadioButton</b> is checked
<b>Text</b>	The text displayed to the right of the <b>CheckBox</b> or <b>RadioButton</b>
<b>CheckedChanged()</b>	Default event that is generated when the <b>Checked</b> property changes

**Table 12-6** Commonly used **CheckBox** and **RadioButton** properties and default event



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



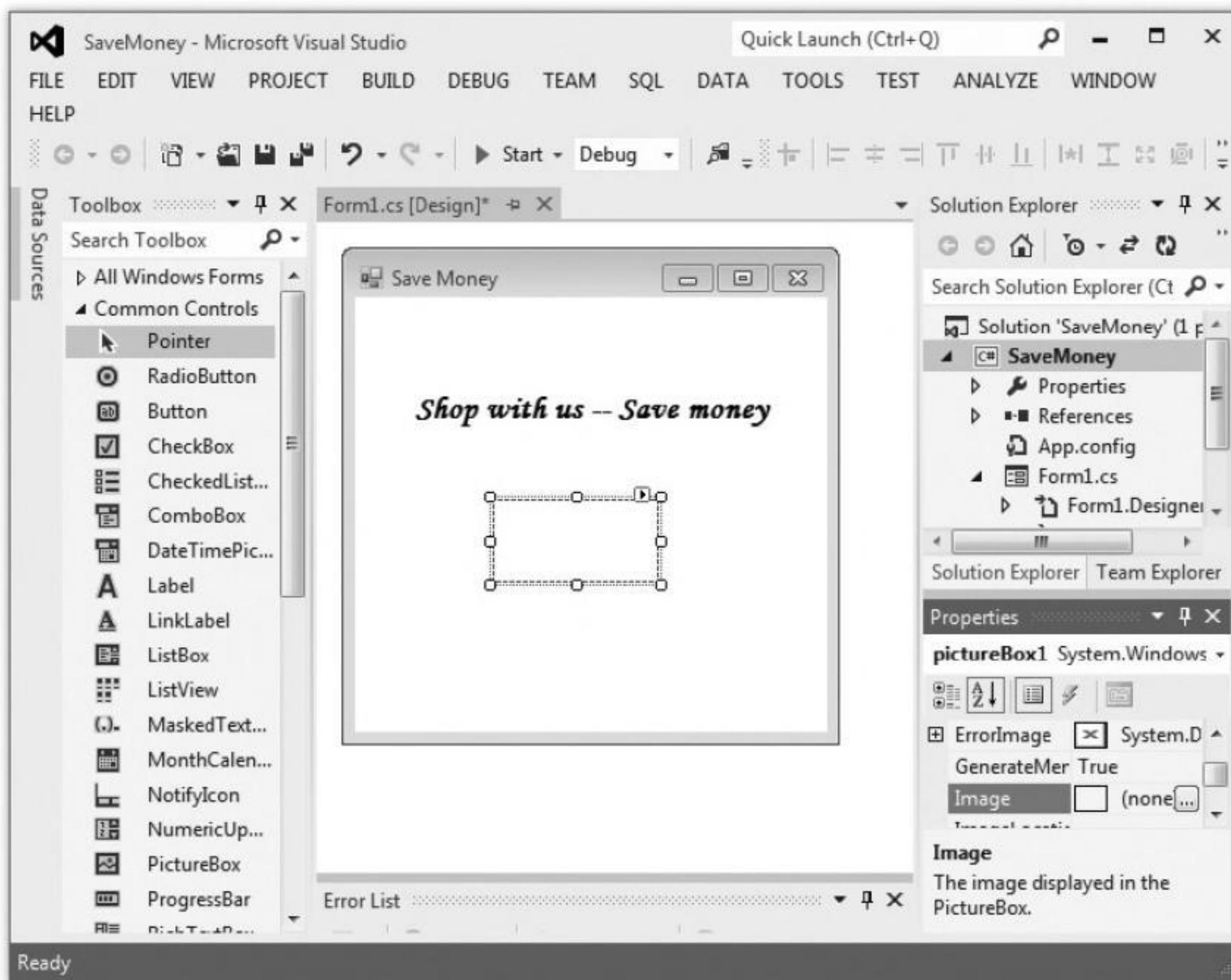
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Figure 12-18 shows a new project in the IDE. The following tasks have been completed:

- A project was started.
- The Form Text property was changed to “Save Money”.
- The Form BackColor property was changed to White.
- A Label was dragged onto the Form, and its Text and Font were changed.
- A PictureBox was dragged onto the Form.



**Figure 12-18** The IDE with a Form that contains a PictureBox

In Figure 12-18, in the Properties list at the right of the screen, the Image property is set to *none*. If you click the button with the ellipsis, a Select Resource window appears, as shown on the left in Figure 12-19. When you click the Import button, you can browse for stored images. When you select one, you see a preview in the Select Resource window, as shown on the right in Figure 12-19.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

in the list. The method returns `true` if an item is selected and `false` if it is not. Therefore, code like the following could be used to count the number of selections a user makes from `majorListBox`:

592

```
int count = 0;
for(int x = 0; x < majorListBox.Items.Count; ++x)
    if(majorListBox.GetSelected(x))
        ++count;
```

Recall that the first position in an array is position 0. The same is true in a `ListBox`.



Alternatively, you can use a `ListBox`'s `SelectedItems` property; it contains the items selected in a list. The following code assigns the number of selections a user makes from the `majorListBox` to count:

```
count = majorListBox.SelectedItems.Count;
```

The `SelectSelected()` method can be used to set a `ListBox` item to be automatically selected. For example, the following statement causes the first item in `majorListBox` to be the selected one:

```
majorListBox.setSelected(0, true);
```

A **ComboBox** is similar to a `ListBox`, except that it displays an additional editing field to allow the user to select from the list or to enter new text. The default `ComboBox` displays an editing field with a hidden list box. The application in Figure 12-25 contains a `ComboBox` for selecting an airline. A **CheckedListBox** is also similar to a `ListBox`, with check boxes appearing to the left of each desired item. The application in Figure 12-25 uses a `CheckedListBox` for flight options.



Figure 12-25 The `FlightDemo` application after the user has made some selections



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



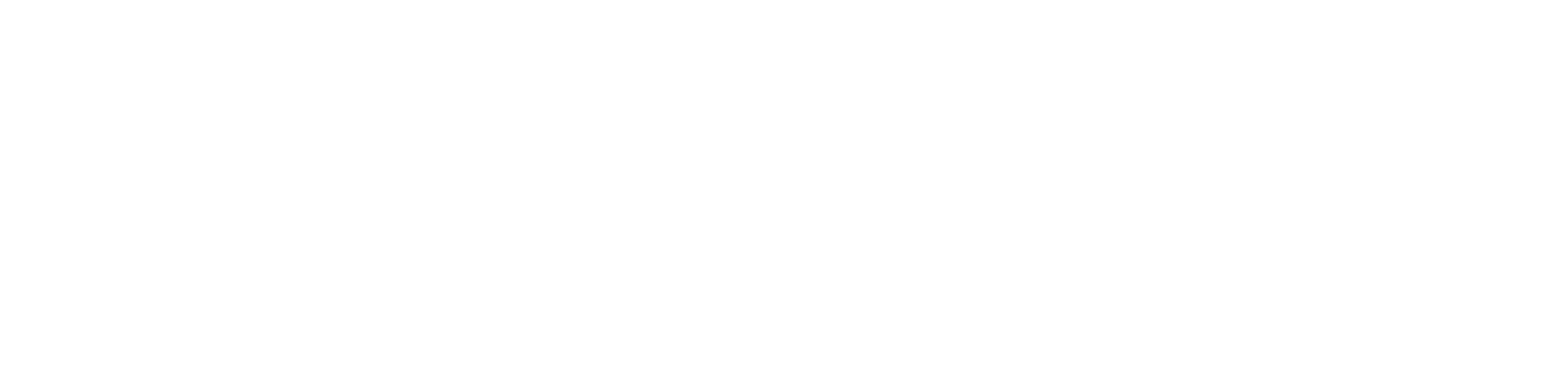
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

(continued)

```
class DiscountDelegateDemo
{
    public static void StandardDiscount(ref double saleAmount)
    {
        const double DISCOUNT_RATE = 0.05;
        const double CUTOFF = 1000.00;
        double discount;
        if(saleAmount >= CUTOFF)
            discount = saleAmount * DISCOUNT_RATE;
        else
            discount = 0;
        saleAmount -= discount;
    }
}
```

633

3. Add a PreferredDiscount() method. The method also accepts a reference parameter that represents the amount of a sale and calculates a discount of 10 percent on every sale.

```
public static void PreferredDiscount(ref double saleAmount)
{
    const double SPECIAL_DISCOUNT = 0.10;
    double discount = saleAmount * SPECIAL_DISCOUNT;
    saleAmount -= discount;
}
```

4. Start a Main() method that declares variables whose values (a sale amount and a code) will be supplied by the user. Declare two DiscountDelegate objects named firstDel and secondDel. Assign a reference to the StandardDiscount() method to one DiscountDelegate object and a reference to the PreferredDiscount() method to the other DiscountDelegate object.

```
static void Main()
{
    double saleAmount;
    char code;
    DiscountDelegate firstDel, secondDel;
    firstDel = new DiscountDelegate(StandardDiscount);
    secondDel = new DiscountDelegate(PreferredDiscount);
```

5. Continue the Main() method with prompts to the user to enter a sale amount and a code indicating whether the standard or preferred discount should apply. Then, depending on the code, use the appropriate delegate to calculate the correct new value for saleAmount. Display the value and add closing curly braces for the Main() method and the class.

(continues)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

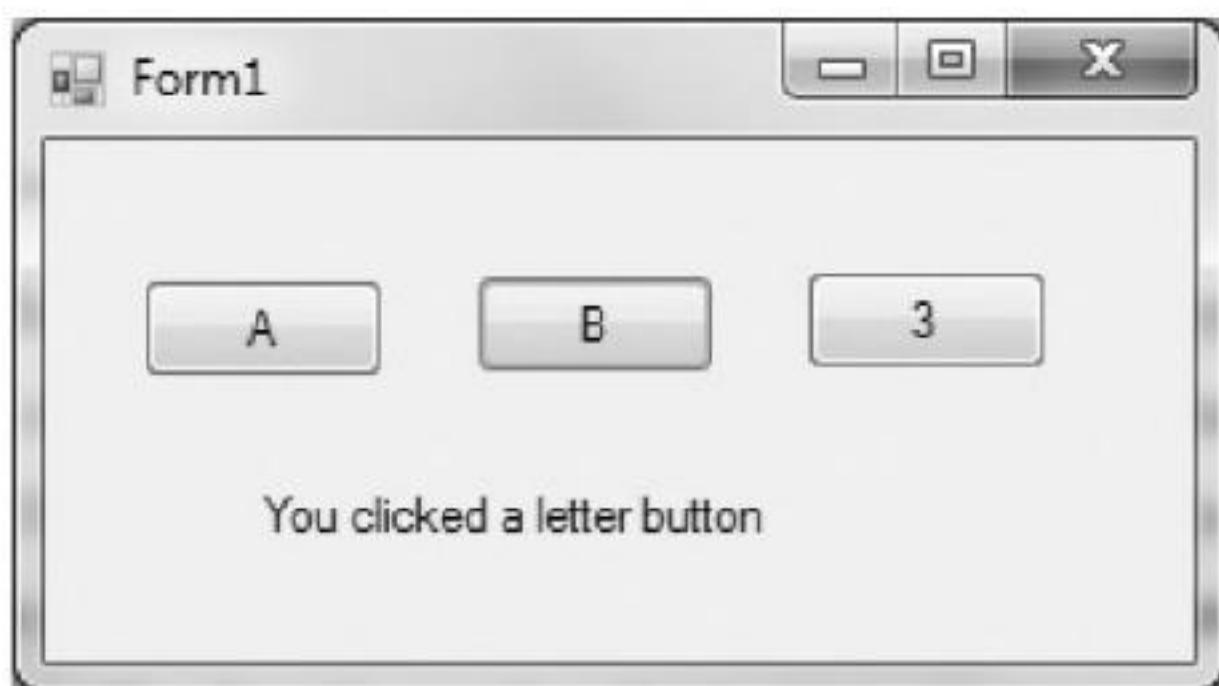


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

If you click the second button so its properties are displayed in the IDE's Properties list, you can click the Events icon to see a list of events associated with the second button. If no Click event has been chosen yet, a list box is available. This list contains all the existing events that have the correct signature to be the event handler for the event. Because the `button1_Click()` handler can also handle a `button2_Click` event, you can select it as the event for `button2`. When you run the program, clicking either button produces the output shown in Figure 13-26.



**Figure 13-26** Output of the `SingleHandler` program after either letter button is clicked

Until you associate an event with the “3” button, nothing happens when the user clicks it. To complete the application that is running in Figure 13-26, you would want to associate an event with the “3” button to modify the Label’s Text to “You clicked a number button”.



When two or more `Controls` generate the same event, many programmers prefer to generalize the event method name. For example, if `button1` and `button2` call the same method when clicked, it makes sense to name the event method `letterButton_Click()` instead of `button1_Click()`.

Perhaps you have shopped online at a site that offers multiple ways to “buy now.” For example, you might click a grocery cart icon, choose “Place order now” from a menu, or click a button. Often, “Place order now” buttons are displayed at several locations on the page. If you want to encourage a user’s behavior, you should provide multiple ways to accommodate it.



Watch the video *Managing Multiple Controls*.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



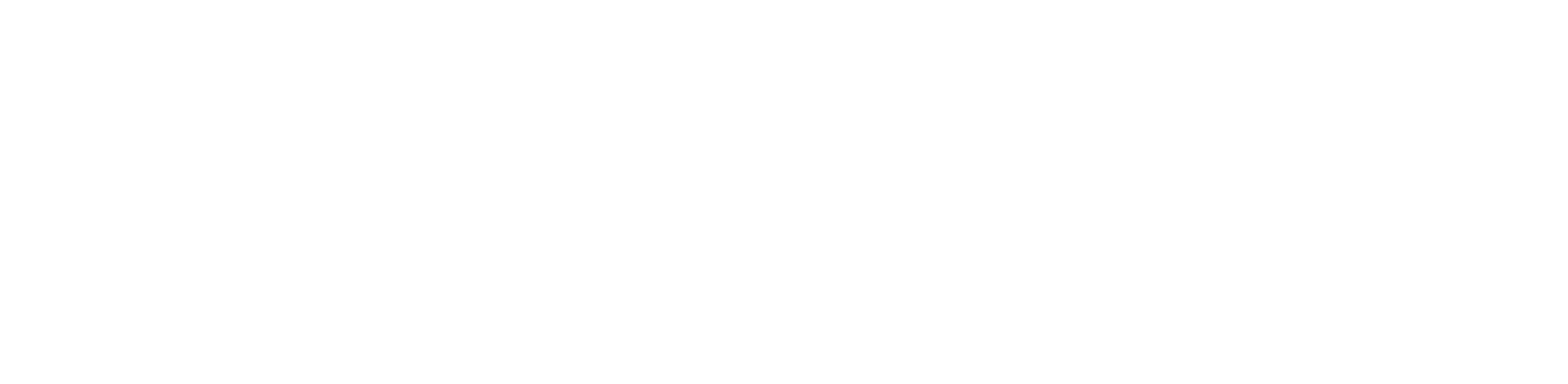
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



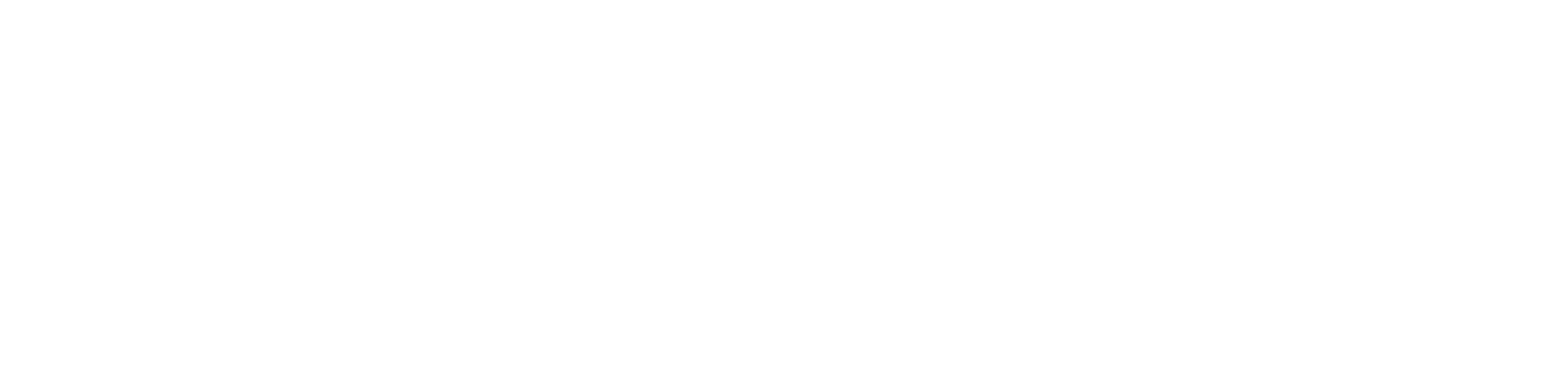
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



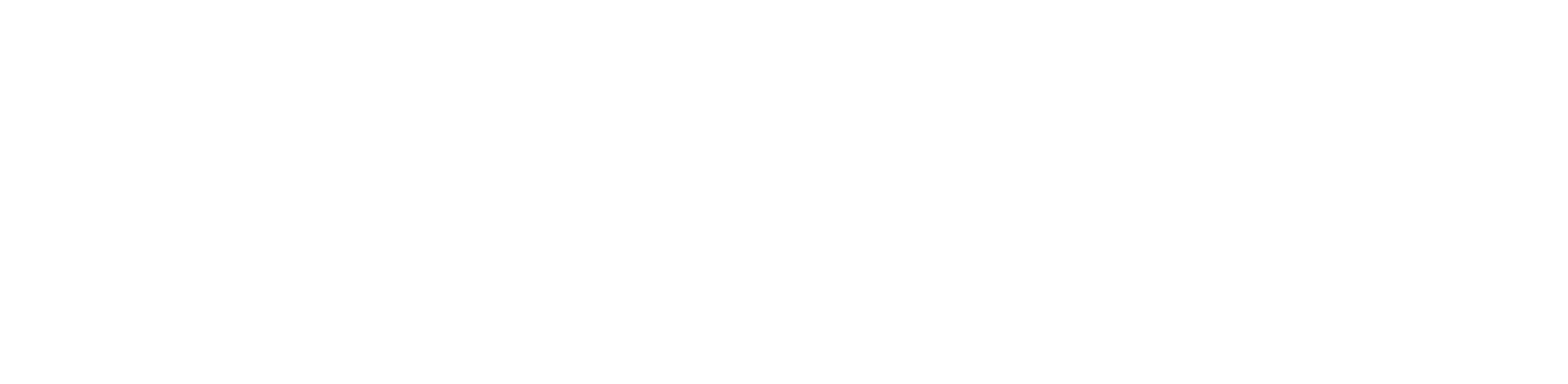
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



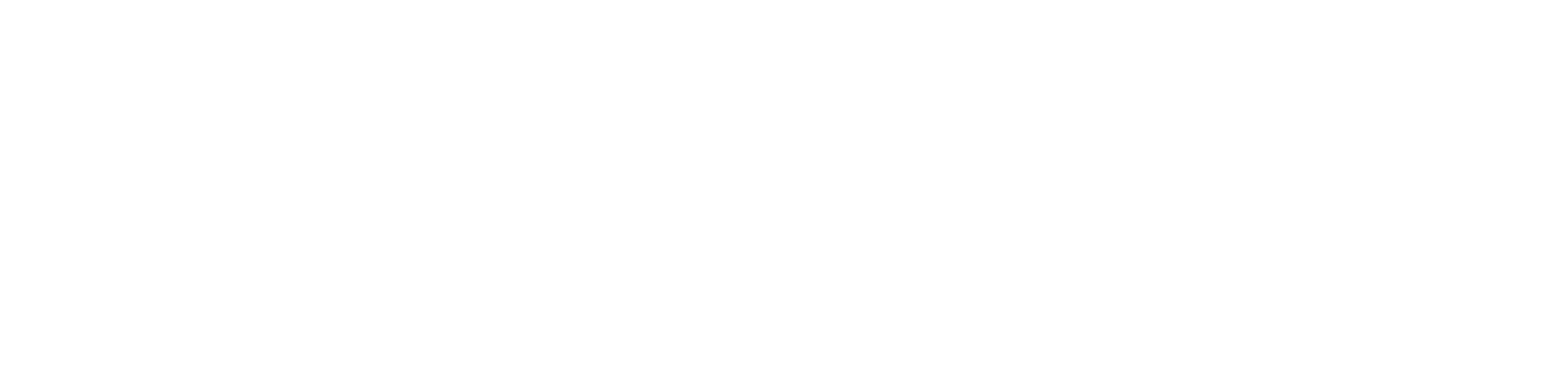
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



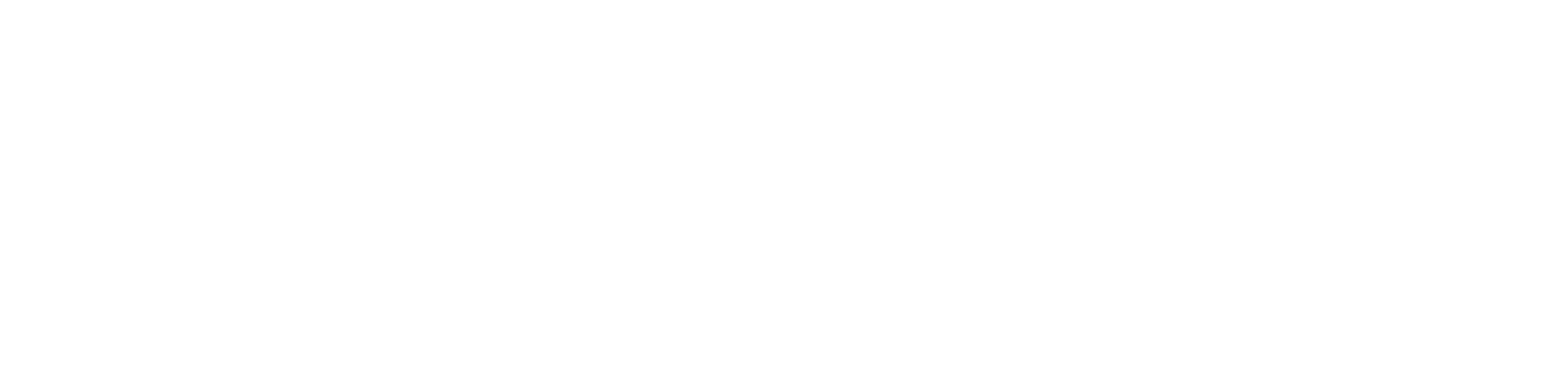
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



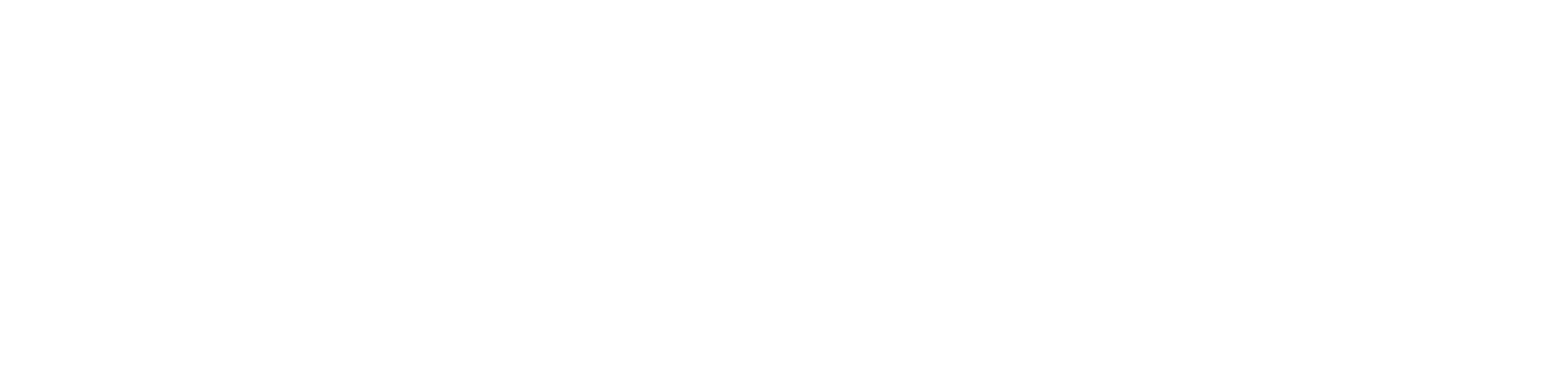
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



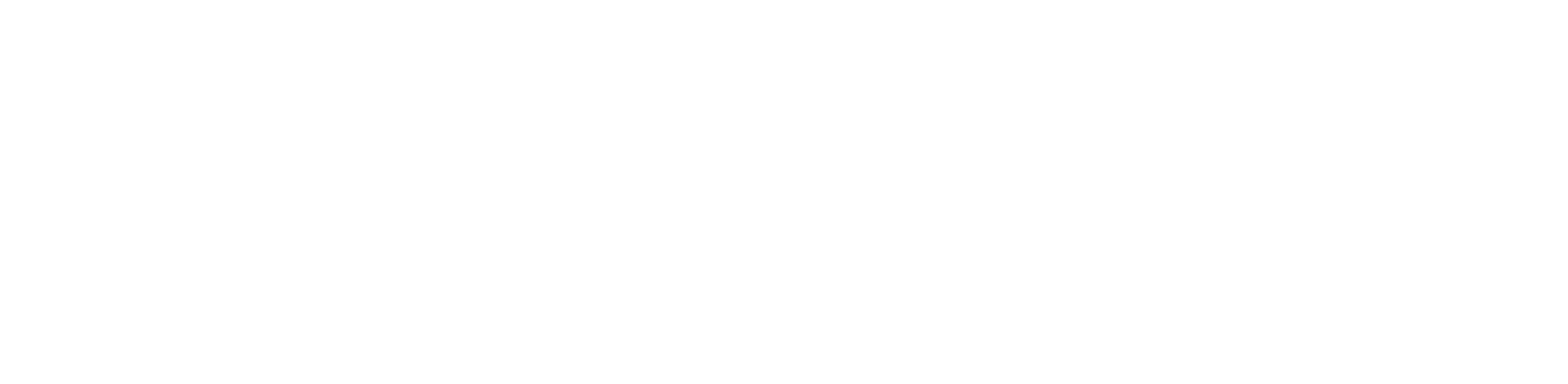
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

# Index

All bold page numbers indicate definitions

## A

aBackspaceChar variable, 77  
AbeLincoln file, 610  
abstract classes, **475**–478, 482–483, 492  
abstract keyword, 475, 476  
abstract methods, **476**, 477, 479–483, 492  
AcceptButton property, 108  
Access databases  
    automatically assigning numbers to records, 739  
    creation, 737–739  
    Datasheet view, 739  
    designing table, 738  
    retrieving data from, 747–750  
    tables and LINQ queries, 755–759  
access keys, 582  
access modifiers, 372  
    private and public, 389–392  
    properties, 381  
access specifiers, **114**, 136  
AccessCars program, 752–754, 760  
accessibility, **287**–288, 319  
accessibility modifiers, 639  
accessors, **380**, 427  
AccessSomeNames program, 700–703  
AccountNum property, 543  
AccountNumber object, 485  
accumulated, **219**, 231  
accumulating totals, 219–221  
actions if only one condition true, 162  
ActiveLinkColor property, 570  
actual parameters, **297**, 319  
add and assign operator (**+=**), **67**, 94, 630, 639  
Add Connection dialog box, 752  
add identifier, 15  
Add New Item window, 605  
AddBooks program, 410  
AddCheckDigit() method, 485  
AddDays() method, 594

addition operator (+), 65, 407, 409, 410  
AddMonths() method, 594  
AddRangeObjectsDemo project, 589  
AddYears() method, 594  
AdvertisingMessage() method, 393–394  
age variable, 161  
ageOverMinimum variable, 175  
aggregate operators, **759**, 776  
alert (\a) escape sequence, 77  
alias identifier, 15, **48**–49  
aliases, 92, **330**, 331, 360  
ambiguous methods, **349**–351, 361  
American Standard Code for Information Interchange (ASCII), **789**–792, 796  
AMethod() method, 350–351, 359  
ancestors, **442**, 492  
Anchor property, **598**, 617  
AND operator, **160**, 186, 736  
anExceptionInstance object, 512–517  
Animal class, 442, 476, 477  
Animal constructor, 477  
annualIncome variable, 72  
annualSalary variable, 58  
anomalies, **734**, 775  
answer variable, 516  
AnyLegalClassName identifier, 16  
Append mode, 684, 685  
applicable methods, **345**, 361  
application classes, **13**, 40  
application software, **2**, 38  
ApplicationException class, 506, 540–541  
applications  
    mission-critical, 507  
    terminating, 529  
Area property, 391–392  
args array, 313  
args variable, 313  
arguments, **11**, 40, 284  
    constructors, 472–473

default, 301  
delegates, 638  
methods, 289–290  
named, 354–359  
positional, **354**  
unnamed, 353–354  
arithmetic expressions, 69–70  
    numeric data type, 76  
    parentheses () and, 66  
arithmetic operators  
    binary operators, **64**  
    operands, **64**  
    operator precedence, **65**–66  
    order of operations, 66  
    shortcut, 66–68  
ArithmeticException class, 507  
array elements, **242**, 244, 274  
    accessing, 246–249  
    assigning values, 242–246  
    character fields, 245  
    default values, 245  
    numbering, 243–244  
    numeric fields, 245  
    sequential search, **252**  
ArrayDemo1 program, 250–251  
ArrayDemo2 program, 264–266  
arrays, **242**, 274, 729  
    accessing elements, 246–249  
    ascending order, 245, 261–262  
    BinarySearch() method, 413–417  
    curly braces enclosing values, 245  
    declaring, 242, 412–419  
    elements, 242–243  
    finding value in sorted, 259–261  
    first element in, 244  
    GUI programs, 271–273  
    identifiers, 242–243  
    index out of bounds, 539  
    initializer list, **245**  
    initializing, 245–246  
    jagged, **270**–271

- arrays (*continued*)  
 Length property, 247–248  
 manipulating, 259–266  
 methods, 413  
 multidimensional, 266–271  
 multiple columns, 729  
 naming, 243  
 navigating, 248–249  
 one-dimensional, 266  
 parallel, 252  
 parameter, 330, 336–337  
 passed by reference, 309  
 passing values to methods, 308–312  
 processing elements, 741  
 rectangular, 267  
 reference type, 412  
 resizing, 243, 248  
 reverse order, 262–263, 265  
 searching, 413–417  
 serializable objects, 704  
 single-dimensional, 266  
 sorting, 265, 413–417  
 square brackets ([ ]), 242–243  
 subscript, 244  
 three-dimensional, 269–270  
 two-dimensional, 266–267  
 upper limit of each range, 258  
 value automatically altered, 248  
*Array.Sort()* method, 418  
*Artists* class, 475  
 “as is” classes, 489  
 ASCII. *See American Standard Code for Information Interchange (ASCII)*  
 assembler, 3  
 assemblies, 372, 427  
 assembly language, 3  
 assignment, 50–51, 93  
 assignment operator (=), 50–51, 71, 93, 154, 176, 376, 630  
 get accessors, 381  
 removing delegate, 639  
 right-associative, 785  
 set accessors, 381  
 associativity, 66, 94, 785, 786  
 at runtime, 181, 187, 624, 666  
*aTabChar* variable, 77  
 attributes, 5, 729  
     associating information with code, 704  
     changing, 8  
     child classes, 475  
     classes, 371  
     controls, 560  
     discovering values of, 8  
     objects, 7, 39  
     status, 8  
 auto-implemented accessors, 385  
 auto-implemented properties, 383, 384–385, 389, 427  
 automatic properties, 384, 427  
 automatically generated code, 563–566  
 Average() aggregate operator, 759  
 averaging variables, 242  
*aWorker* identifier, 397  
*aWorker* object, 404
- B**
- BackColor property, 107–108, 112, 133, 581, 586, 588, 606, 611, 664  
 backing fields, 381, 427  
 backslash (\) escape sequence, 77  
 backspace (\b) escape sequence, 77  
 BaileysForm class, 576, 579  
 Balance property, 543  
 balanceDue variable, 67  
 BankAccount class, 541–542, 644–646  
 BankAccountBalanceAdjusted()  
     method, 645, 646  
 bankBal variable, 67, 204, 213  
 BankBalance Form, 229  
 bankBalance variable, 75  
 base 2 numbers, 787  
 base 10 numbers, 787  
 base 16 shorthand, 77–78, 95  
 base 16 system, 793, 796  
 base classes, 441, 483, 491  
     accessing methods and properties, 457–458  
     constructors, 470–475  
     methods, 457  
     overriding members, 453–458  
     size, 441  
     visible members, 456  
 base keyword, 457, 461, 472, 492  
 BASE\_PRICE constant, 583  
 BASIC programming language  
     array elements, 244  
     interpreters, 3  
     STEP keyword, 207  
 BedAndBreakfast form, 605–614  
     namespace header, 579  
     radio buttons, 611–614  
 bedrooms variable, 270  
 Begin value, 697  
 behaviors, 5–6, 39  
 BelleAire CheckBox, 605, 608–609  
 BelleAire form, 608  
 BelleAire Suite CheckBox, 608, 614  
 belleAireCheckBox property, 605  
 belleAireCheckBox\_Checked  
     Changed() method, 608  
 belleAireDescriptionLabel label, 606  
 betterness rules, 345–346, 361  
 bigFont object, 568  
 BillPatients() method, 445  
 bin folder, 35  
 binary files, 674, 687, 719  
 binary form, 407  
 binary numbering system, 788–792, 795  
 binary operators, 64, 94, 408  
     left-associative, 785  
     operands, 173  
     overloadable, 408  
 binary search, 260, 275  
 BinaryFormatter class, 704–705  
 BinaryReader class, 687  
 BinarySearch() method, 259–261, 275  
     arrays, 413–417  
     Sort() method, 262  
 BinarySearchDemo program, 260–261  
 BinaryWriter class, 687  
 BindingSource object, 748, 753, 776  
 bits (binary digits), 787–788, 794, 796  
 bitwise operators, 163, 186  
 black box, 8, 39, 285, 318  
 block comments (/\* \*/), 18, 41  
 blocks, 176, 186  
     curly braces, 151  
     if-else statements, 155  
     if statements, 150–151  
     unreachable, 524  
 Blue button, 664  
 BMethod(12) method, 359  
 bonus parameter, 357  
 Book class, 393, 408–411  
     methods explicitly using this references, 394–395  
 Book parameter, 409, 411  
 Books parameter, 410  
 bool data types, 48, 70–72, 94, 148  
 bool return type, 302  
 Boole, George, 71  
 Boolean columns, 732  
 Boolean expressions, 148  
     evaluating, 198  
     if statement, 147  
     inserting semicolon after, 176  
     logical AND operator (&), 163, 186  
     logical inclusive OR operator (||), 163, 186  
     methods and side effects, 304  
     negating result, 175  
     parentheses (), 161  
 Boolean values, 147  
     methods returning, 304  
     TryParse() method, 335  
 Boolean variables, 70–73, 94, 154, 175  
     if statement, 147

- BorderDemo1 program, 340–341  
 BorderDemo2 program, 343–344  
 Borland Paradox, 738  
 Box class, 405–406  
 break keyword, **169**, 187  
 break statement, 170, 529  
 BreakfastOptionForm form, 611–614  
 BreakfastOptionForm\_Load()  
     method, 612  
 Budget2012 class, 15  
 BUDGET2013 class, 16  
 BudgetForm form, 108  
 Build failed error message, 123  
 Build Solution command, 27, 124  
 built-in  
     methods, 305  
     namespaces, 28  
 BusDriver class, 482–483  
 businesses storing data, 680  
 button1 button, 110–111, 114  
 button2 button, 114, 661  
 Button class, 182, 422–424, 559, 563,  
     565, 650–651, 754, 758, 764  
 Button controls, 117, 599, 611, 624,  
     660, 663, 709, 712, 715  
 button1 object, 424  
 ButtonBase class, 559, 581–584  
 button2\_Click event, 661  
 button1\_Click() method, 114, 118,  
     121–122, 288, 661  
 buttons, 131, 422–424, 558, 563, 709  
     adding, 109–115, 134  
     clicking, 113  
     default event, 113  
     focus, 119  
     functionality, 112–115, 132–133  
     naming, 110, 114  
     properties, 111–112  
     repositioning, 131–132  
     resizing, 111, 131  
     Text property, 756  
 Buttons control, 660  
 byte data type, **57**–**58**, 60, 84, 93, 169  
 bytes, **675**, 704–707, 720, 794, 796
- C**
- C programming language, 4  
     format specifier, 63  
 C# programming language, 2–3, 5,  
     **9**–**10**, 14, 40  
     camel casing, 5  
     case sensitivity, 5, 31  
     exponential operator, 65  
     GUI interface, 10  
     Language INtegrated Query  
         (LINQ) Projects, **743**–**746**  
     methods, 5  
     objects, 10  
     pointers, 10  
     standardization, 10  
     statements ending with  
         semicolon, 11  
     strongly typed, **742**  
 C++ programming language, 2, 3, 5  
     aliases, 50  
     declaring array, 242  
     function pointer, 628  
     Integrated Development  
         Environment (IDE), 104  
     pointers, 10  
     try and catch blocks, 530  
 C# programs  
     namespace-dot-class-dot-  
         method format, 12  
     producing output, 11–13  
     starting point, 12  
 C Sharp compiler, 23  
 CalcArea() method, 392  
 CalcPay() method, 302–304  
 CalcPhoneCallPrice() method,  
     306–307  
 calculatePayroll() method, 627  
 calendar object, 595  
 CalendarForeColor property, 596  
 CalendarMonthBackground  
     property, 596  
 calendars, 593–596  
 call stack, 548  
     tracing Exception objects  
         through, **535**–**540**  
 called, **284**, 318  
 called methods, **285**, 318  
 calling methods, **5**, 12, 39, **285**, 318,  
     356  
 camel casing, **5**, 39, 732  
 CancelButton property, 108  
 candidate keys, 733  
 CanRead property, 684  
 CanSeek property, 684  
 CanWrite property, 684  
 CarLoan class, 446–448, 460–461, 474  
 Carpet class, 390–392  
 carriage return, 691  
 carriage return (\r) escape sequence,  
     77  
 carsButton button, 769  
 CarsGroupByMake project, 769  
 carsListBox list box, 769  
 CartmanCollege database, 748, 755,  
     756  
 cartmanCollegeDataSet object, 748,  
     750  
 case keyword, **169**, 186  
 case label, **169**–**170**, 187  
 case statements, 170  
 casting, 65  
 Cat class, 476–478, 480–481  
 Cat constructor, 477  
 catch, 512–517  
 catch blocks, **512**–**513**, 517, 533, 541,  
     548  
     bypassing, 521  
     displaying unique message, 523  
     “Error!” message, 538  
     Exception argument, 546  
     generic, 523–524  
     multiple, 520–526  
     not executing, 513, 529  
     rethrowing unnamed exception,  
         544  
     statements after, 528–529  
     unreachable, 524  
     WriteLine() method, 539  
 catch keyword, 512–517  
 catch() method, 512–517  
 catching  
     exceptions, 512–517  
     multiple exceptions, 520–525  
 cd command, 31  
 changeButton button, 624  
 changeButton event handler, 626–627  
 changeButton\_Click() method, 625,  
     627  
 Changed() event, 639–640  
 Changed field, 640  
 ChangedEventHandler delegate, 638  
 ChangedEventHandler identifier, 638  
 changeLabel method, 627  
 changeOutputButton button, 134  
 changeOutputButton\_Click()  
     method, 134  
 Chapter.01 folder, 31–32, 35  
 char data type, 48, **57**–**58**, 76–78,  
     93–94, 169  
 char variable, 77  
 character arrays, 80  
 character fields, 245  
 character set, **680**, 720  
 characters, 57–58, 76–78, **680**–**681**,  
     720  
     bits (binary digits), 787–788  
     case sensitivity, 204  
     nonprinting, 77  
     numbers as, 76  
     strings, 76  
 char.TryParse() method, 336  
 check boxes, 581–584, 605–614  
 check digits, **485**, 493  
 CheckBox class, 559, 581–584,  
     605–614, 616  
 checkBox1 object, 582

- checkBox2 object, 582  
 checkboxes, 582  
**C**heckBoxes objects, **581**  
 checked list box, 592  
 Checked property, 581–583  
 CheckedChanged() method, 581, 583, 608  
**C**heckedListBox, **592**, 617  
 CheckedListBox objects, 588–592  
 Child class, 482  
 child classes, **441**, 491  
     accessing parent class data, 449  
     attributes, 475  
     inheritance, 445, 465  
 Choose a Data Source Type dialog box, 752  
 Choose Your Data Connection dialog box, 752  
 CivilWarSoldier class, 483  
 class body, 373  
 class clients, **371**, 426  
 class definition, **372**, 426  
 class diagrams, 440  
 class fields, 373  
 class header, 14, 21, **372**, 426  
 class keyword, 16, 372  
 class users, **371**, 426  
 class-wide methods, 374  
 classes, 6–7, 39, 370, 375  
     abstract, 475–478  
     access modifiers, **372**–**373**, 426  
     “as is,” 489  
     class body, 373  
     class definition, **372**  
     class header, **372**  
     Common Language Runtime exception, 506  
     comparisons between objects, 413  
     concepts, 370–371  
     concrete, **475**  
     constructors, 377, **396**–**403**  
     creation, 14, 386–389  
     data, 371–372  
     default constructor, 405  
     derived, 489  
     destroying instances, 420–422  
     extending, 443–445, 447  
     fields, 375  
     fragile, **452**  
     grouping, 12  
     illegal names, 16  
     implementing interface, 414  
     inheritance, **8**, **438**–**442**, 475  
     instance variables, 373–375  
     instances, **7**, **370**, 414  
     instantiating objects, 370, 372–375  
     Main() method, 13  
     methods, **284**, 290–292, 372–375, 482–483  
     named constants, 391  
     names, 8, 14, 16, 372  
     namespaces, 19–20, 466  
     objects, 7–8, 466  
     overloaded constructors, 402–403  
     overloaded operators, 408  
     parameterless constructors, 398–399  
     private data fields, 373–374  
     runnable, 13  
     sealed, **484**  
     unconventional though legal names, 16  
     upper camel casing, 8  
     user-defined application exception, 506  
     valid and conventional names, 15  
 clerk object, 445, 448  
 Click delegate, 653  
 Click() event, **114**, 136, 315, 570, **627**, 650–651, 655, 660, 661, 664, 666  
 Click for meal options button, 614  
 Click() method, 113, 115, 127, 134, 180, 182, 284, 316, 585, 603, 613, 650–651, 750, 758, 766, 770  
 clickLocationLabel label, 656  
 ClickParameterDemo program, 315  
 clicks and x- and y-coordinate position, 656  
 clients, **285**, 318  
 Close() method, 715  
 closing files, **682**, 720  
 Closing() method, 355–356  
 COBOL array elements, 244  
 code  
     automatically generated, 563–566  
     comments, 18  
     expanding or condensing, 106  
     line numbers, 124  
     side effects, **163**  
 code bloat, **286**, 318  
 Code Editor, 797, 799–800  
 code refactoring, **122**, 136  
 code snippets, **799**, 800  
 code window, 112, 124  
 CodeDomSerializerException class, 504  
 collections, 775  
 Color class, **573**–**575**, 616  
 Color field, 751  
 colors and forms, 573–575  
 columns, 729, 732–733  
 combo boxes, 592  
 ComboBox class, **592**, 617  
 ComissionEmployee class, 443  
 comma delimiters, 688  
 command line, **23**–**24**, 41  
     compiling and executing programs, 31–32  
 command prompt, **23**, 23–25, 41  
     paths, 31  
 comment out, **18**, 40  
 comments, 17–18, 36–37, 565  
 CommissionEmployee class, 439–441, 443–444, 449–451, 457, 463, 470–473  
 CommissionEmployees class, 452  
 commissionRate field, 439, 443, 449  
 CommissionRate property, 449–450, 452  
 commissionRate variable, 177–178  
 Common Language Runtime exception classes, 506  
 ComparableEmployeeArray program, 416  
 Compare() method, 80, **81**, 82, 95  
 CompareNames1 program, 80  
 CompareThreeNumbers project, 157–159  
 CompareTo() method, 80, **81**, 82, 95, **413**–**417**, 419, 428  
 comparing  
     objects, 467–468  
     String objects, 468  
     strings, 80–82  
     to zero, 223–224  
 comparison operator (==), 80, 94, 148, 176  
 compilers, **3**, 24–25, 38  
 compiling programs  
     command line, 31–32  
     command prompt, 23–25  
     intermediate version, 22  
     Visual Studio IDE, 26–28, 32–35  
 completion lists, 797–798  
 completion mode, **797**, 800  
 complex data types, 704  
 Component class, 488, **558**, 616  
 components and focus, 119  
 composed delegates, 627, 634–637, 666  
 composite keys, **729**, 775  
 composition, **375**, 427  
 compound Boolean expressions, 160–161  
 compound conditions, 208  
 compound expressions  
     Boolean logical AND operator (&), **163**  
     Boolean logical inclusive OR operator (||), **163**

- combining conditional AND  
(`&&`) and OR (`||`) operators, 163–165
- conditional AND operator (`&&`), **160**–161, 166–168
- conditional OR operator (`||`), **162**, 166–168
- if statements, 160–165, 167
- parentheses to correct logic, 164–165
- side effects, **163**
- compound keys, **729**, 775
- `ComputeGross()` method, 357–358
- `ComputePaycheck()` method, 296
- computer files, **674**, 719
- computer simulations, **6**, 39
- computers, 2
- `ComputeTotalPay()` method, 357–358
- `conBreakfastButton` RadioButton control, 613
- concatenate, **53**, 93
- concatenating strings, 56–57, 407
- concrete classes, **475**, 492
- conditional AND operator (`&&`), **160**–161, 166–168, 186
- appropriate use of, 178–179
- combining with conditional OR operator (`||`), 163–165
- truth tables, 160
- conditional operator (?), **173**–174, 187
- conditional OR operator (`||`), **162**, 166–168, 186
- appropriate use of, 178–179
- combining with conditional AND operator (`&&`), 163–165
- if statements, 162
- Connelly, Marc, 558
- console applications, 113, 127
- `Console` class, 12, 19, 288, 489, 683
- Console Project, 26
- `ConsoleApplication1` project, 32
- `Console.Error` stream, 683
- `Console.In` stream, 683
- `Console.Out` stream, 683, 685
- `Console.Read()` method, 87
- `Console.ReadLine()` method, **86**–87, 95, 305
- `Console.WriteLine()` method, 54
- `Console.WriteLine()` method, 54, 305, 345, 755
- const field, 402
- const keyword, 83, 391
- const modifier, 402
- constants, **48**, 50, 92, 713
- assigning value, 84, 171, 402
- data types, **48**–49
- declaring in methods, 292
- floating-point, 61, 568
- numeric, 568
- public, 391
- represented by identifiers, 84–86
- constructor initializers, **400**–**401**, 428
- constructors, 424, 427
- automatically supplying, 397, 472
- base classes, 470–475
- default, 377, **397**–398
- derived classes, 470, 474–475
- initializing fields, 397
- nondefault, 412
- overloading, 398–400, 402–403
- parameterless, **398**–399
- passing parameters to, 397–398
- readonly modifier, 402
- requiring arguments, 472–473
- throwing exceptions, 543
- two-parameter version, 401
- writing, 397
- ContainerControl class, 488
- ContainerControls, 600
- Contains() method, 758
- `contBreakfastButton` button, 611
- contextual keywords, **381**, 427
- Control class, 488, **558**–562, 588, 616, 659
- control statement, **150**, 186
- controls, **104**–105, 136, 558–562, 604
- acting in expected ways, 114
- adding to forms, 109–110
- aligning, 120, 597
- anchoring, 598
- attributes, 560
- camel casing, 121
- `CheckedListBox` objects, 588–592
- `ComboBox` objects, 588–592
- containers, 600
- `DateTimePicker` class, **593**–596
- default event, **570**
- default names, 121
- deleting, 110
- docking, 599
- dragging, 597
- events, 648–653
- fixed distance from side of container, 598
- focus, **119**–120, **659**–660, 663–664
- Font property, 120
- fonts, 566–569, 607
- generating same event, 661
- `ListBox` objects, 588–592
- managing multiple, 659–661
- `MonthCalendar` class, **593**–596
- mouse events, 654–656
- moving, 110
- naming, 121–122
- `PictureBox` object, **585**
- position, 559
- properties, 112
- protected properties, 560
- public properties, 560
- renaming, 121–122
- size, 559
- tab order, **119**–120, 659
- `TabIndex` property, 663–664
- Controls class, 565
- ConversionWithParse program, 334
- ConversionWithTryParse program, 335
- Convert class, 87–88, 333
- ConvertEmployeeToString() method, 466
- converting strings, 87
- Convert.ToInt32() method, 346–347, 531
- Coordinated Universal Time (UTC), 676
- Count() aggregate operator, 759, 765
- CountContributions program, 272–273
- counted loops, **202**, 231
- counter variable, 66
- Create() method, 676
- Create mode, 684–685
- CreateDirectory() method, 678
- CreateEmployee class, 377
- CreateEmployee program, 377
- CreateEmployee2 program, 382
- CreateNameFile program, 693–696
- CreateNew mode, 685
- CreateSomeEmployees class, 399–400
- CreateStudent.cs file, 387
- CreateStudents class, 403
- CreateStudents2 class, 403
- CreateStudents3 class, 418
- CreateText() method, 676
- CreateTwoEmployees program, 378
- credits field, 453–456
- Credits property, 453, 455–456
- .cs extension, 24
- csc command, 23
- csc.exe file, 24
- CSV files, **688**, 721
- CultureInfoClass, 64
- cultures, **64**, 94
- curly braces `{}`, 14
- currency values, 63–64, 732
- Current value, 697
- custId column, 736
- custom Exception class, 540–543
- customerID column, 733
- CustomFormat property, 596



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- EnterInvoices program, 708–712  
 entities, 729  
 Enum class, 249  
 enum constants, 84–86  
 enum keyword, 84  
 enumeration data type, 86, 169  
 enumerations, 84–86, 95  
     displaying values, 249  
     GUI applications, 183–184  
     switch statements, 171–172  
 Environment.Exit() method, 529  
 Equal to (==) operator, 71  
 Equals() method, 80, 82, 95, 466  
     overloading, 467  
     overriding, 467–468  
 error-handling, traditional methods  
     for, 511–517  
 “Error in the application” error  
     message, 541  
 error lists, 105, 124, 130, 136  
 “Error!” message, 538  
 error messages, 23  
     descriptive, 517–519  
     viewing, 124  
 error report window, 509  
 errors, 504–507  
     *See also* exceptions  
     dividing by zero, 511  
     execution-time, 504  
     infrequent events, 511  
     runtime, 504  
 escape sequences, 77, 79, 95  
 event driven, 624  
 event-driven programs, 181, 228, 666  
 event handlers, 624, 666  
     declaring your own, 638–641  
     handling multiple events, 660–661  
     naming, 625  
 event handling, 624–627  
 event-handling methods, 125  
 event listener creation, 645–647  
 event receivers, 625, 666  
 event senders, 625, 666  
 event wiring, 627, 666  
 EventArgs argument, 638  
 EventArgs class, 626, 638–639, 645,  
     654–655, 666  
 EventArgs.Empty argument, 639  
 EventDemo application, 625–626  
 EventHandler delegate, 641–643, 666  
 EventListener class, 640–643, 645–647  
 events, 114, 136, 370, 624, 666  
     accessibility modifiers, 639  
     associating method with multiple,  
         664–665  
     controls, 648–653, 661  
     declaring, 638–641  
     default, 570  
     determined by user’s actions,  
         181–182  
     firing, 624  
     GUI programs, 624  
     infrequent, 511  
     invoking, 639, 645  
     listing, 652  
     raising, 624  
     runtime occurrences, 624  
     triggering, 624  
     values, 114  
 Events button, 664  
 events-driven programs, 181–182  
 Evergreen class, 441, 442  
 Exabytes (EB), 795  
 Exception class, 504, 506–507,  
     512–517, 519, 540–541  
     custom, 540–543  
     Message property, 517–519  
     subclasses, 504–505  
     ToString() method, 517  
 exception handling, 504, 511,  
     516–517, 547, 686  
     catch, 512–517  
     critical, 511  
     object-oriented methods, 512–517  
     throw, 512–517, 531–534  
     try, 512–517  
 Exception objects, 515–516, 522, 532  
     automatically created, 513  
     information about, 519, 536  
     preventing creation, 511  
     tracing through call stack, 535–540  
 Exception types, 512–517, 524, 540, 543  
 exceptional data, 540  
 exceptions, 504–507, 547  
     catching, 512–517  
     clean-up tasks, 528  
     managing, 504  
     multiple, 520–525  
     naming, 512–517  
     propagating, 536  
     purposely generating, 506–510  
     rethrowing, 544–546  
     terminating programs, 507  
     throwing, 512–517, 521, 531, 543  
     unhandled, 509, 529  
 ExceptionsOnPurpose2 class, 516  
 ExceptionsOnPurpose program,  
     508–510, 516  
 ExceptionsOnPurpose3 program,  
     525–526  
 .exe extension, 25  
 executing programs, 22–37  
     command line, 31–32  
     Visual Studio IDE, 32–35  
 execution-time errors, 504  
 Exists() method, 676, 678  
 explicit casts, 75, 94  
 exponential operator, 65  
 exposes, 720  
 Extended Binary Coded Decimal  
     Interchange Code (EBCDIC),  
     789, 796  
 extended class, 441, 491  
 extending classes, 443–445, 447  
 eXtensible Markup Language  
     (XML), 18, 687, 720  
 extension methods, 484–486, 493

**F**

- F format specifier, 63  
 F1 (Search) key, 126  
 false return type, 302  
 false value, 147–149  
 fault-tolerant programs, 507, 547  
 fields, 316, 371, 426, 681, 720, 728  
     abstract classes, 475  
     altering states, 379–380  
     assigning values, 639  
     initializing, 397  
     left-aligned contents, 388  
     null values, 732  
     objects of other classes, 375  
     private, 390  
     protected, 450  
     public, 390  
     static, 391  
 fields array, 691  
 File class, 675–677, 696, 702, 720  
 file extension, 674  
 file pointer, repositioning, 700–703  
 file position pointer, 697, 721  
 FileAccess enumeration, 685  
 FileMaker Pro, 738  
 FileMode enumeration, 685  
 filenames, 713  
     @ sign, 710  
     misspelling, 23  
 files, 681  
     accessing information about,  
         672–677  
     binary, 674, 687  
     closing, 529–530, 682, 720  
     creation, 674, 693–696  
     CSV, 688  
     data, 674, 680–682  
     directories, 675  
     file extension, 674  
     folders, 675  
     managing, 105  
     modification time, 674  
     names, 674

- opening, **682**, 720  
 random access, **682**  
 reading from, **675**, 684, 695–696, 720  
 related records, 728  
 sequential access, **681**–**682**  
 sizes, 675  
 text, **674**  
 viewing, 105  
 writing to, **675**, 684, 720  
 XML, 687
- FileStatistics program, 677
- FileStream class, 684–685, 688, 690, 693–695, 705, 710, 713
- Fill() method, 750, 754
- finally blocks, 512–517, **528**–**530**, 548
- FindEmployees program, 698–699
- FindPriceWithForLoop program, 253
- FindPriceWithWhileLoop program, 256
- fingerprint, 469
- firing event, **624**, 666
- FirstClass class, 12, 14–16
- FirstClass program, 11
- firstDel delegate, 629, 631, 633, 635
- firstName column, 734
- firstName field, 728, 738
- firstName string, 79
- fixed-pitch font, **120**, 136
- flexibility, 729
- FlightDemo application, 592
- float argument, 568
- float data type, **61**, 93
- floating decimal point, 61
- floating-point addition, 407
- floating-point constants, 568
- floating-point data types, **61**–**64**
- floating-point numbers, **61**–**65**, 93  
     divide by zero, 506
- floating-point variables, 62–63
- floor variable, 270
- flowcharts, **144**–**145**, 185
- focus, **119**–**120**, 136, **659**–**660**, 663–664, 666
- folders, **675**, 678–679, 720
- Font class, **566**–**569**, 616
- Font constructor, 568
- Font dialog box, 578
- Font object, 569
- Font property, 112, 120, 133, 563, 565–567, 578, 586, 607, 609, 611, 624, 650
- Font window, 566–567
- fonts, 563  
     changing, 120–121
- controls, 607
- default, 568, 578  
     selecting, 566–569
- FontStyle enumeration, 84, 568–569
- for keyword, 207
- for loops, **206**–**212**, 227, 231  
     block of statements, 207  
     blocking body, 209  
     breaking out early, 254–255  
     declaring new variable, 208  
     initializing variables, 208  
     nested, 218  
     parentheses, 207  
     performing multiple tasks, 208  
     search arrays, 252–254  
     single statement, 207  
     tasks, 208
- foreach loops, **248**–**249**, 275, 741, 744, 756  
     nested, 766, 770
- ForeColor property, 107–108, 112, 133
- foreign alphabets, 78
- Form class, 121, 228, 422, 487–788, 565, 576, 709
- BackColor property, 586
- default event, 570
- deleting automatically added grid from, 755
- properties, 107, 576
- resizing, 760
- Size property, 624
- Text property, 586, 752, 769
- Form1 class, 487, 565, 663–664
- Form1() constructor, 709
- Form Designer, **105**, 112, 136  
     expanding code, 563  
     folder tab, 107
- form feed (\f) escape sequence, 77
- Form1 form, 107
- Form1 object, 712
- formal parameters, **297**, 319, 328
- Format property, 596
- format specifiers, **62**–**63**, 93
- format strings, **53**–**56**, 93, 296–297
- FormatException exception, 525, 527, 531
- formatting  
     data in GUI applications, 120  
     strings, 54–56
- Form.cs file, 583
- Form1.cs file, 106, 610, 613, 652, 658, 762
- Form1.Designer.cs file, 106, 423, 563, 580, 715
- Form1\_KeyUp() method, 658
- Form1\_Load() method, 750, 754
- Form1.resx file, 106
- forms, **104**, 135, 558
- buttons, 109–115, 131, 134, 422–424, 563, 709
- camel casing, 121
- check boxes, 582, 605–614
- CheckedListBox objects, 588–592
- controls, **104**–**105**, 109–110, 558–562
- creation, 104–109
- dates, 593–596
- default names, 121
- failing to close, 126
- fonts, 563
- Integrated Development Environment (IDE) creation, 563–566
- labels, 115–121, 132–133, 575–582, 598, 709
- layout, 597–600
- list boxes, 588–592
- menu strip, **601**–**604**
- multiple controls, 659–661
- naming, 108, 121–122
- nonstatic methods, 316
- not resizable, 600
- PictureBox objects, 585–587
- printed text appearance, 566–569
- properties, 129
- radio buttons, 582, 611–614
- reading data from text file into, 712–718
- renaming, 130
- saving, 132
- snap lines, **597**
- text, 107, 563
- text boxes, 115–121, 709
- time, 593–596
- title bar, 107
- Forms applications, 104
- formsComboBox objects, 588–592
- FormWithALabelAndAButton program, 563–564
- forward declarations, 10
- fragile classes, **452**, 492
- Frame class, 608
- free-graphics.com, 610
- from clause, 756
- from keyword, **744**, 776
- from-where-select combination, 756
- fullBreakfastButton button, 611, 613
- fully qualified, 319
- fully qualified type name, 50
- function pointer, 628
- functions, 5

**G**

gallonsOfGas value, 511, 513, 517  
garbage, **221**, 231  
GB. *See* Gigabytes (GB)  
generic catch blocks, 523–524  
get accessors, **380**–384, 389, 427, 443, 460  
get contextual keyword, 381  
get identifier, 15  
GetCreationTime() method, 676, 678  
GetData() method, 418  
GetDirectories() method, 678  
GetEmployeeIdentification()  
method, 466  
GetFiles() method, 678  
GetGreeting() method, 443, 457–458  
GetHashCode() method, 466, 469–470  
GetLastAccessTime() method, 676, 678  
GetLastWriteTime() method, 676, 678  
GetLength() method, 312  
GetNames() method, 249  
GetObjectData() method, 704  
GetPrice() method, 304  
GetSelected() method, 591–592  
getter, **381**, 427  
GetType() method, 466, 519  
gigabytes (GB), **675**, 720, 794–795  
global identifier, 15  
GMT. *See* Greenwich Mean Time (GMT)  
Goodbye() method, 629, 631  
goodStudents collection, 755–756  
governing type, **169**, 187  
gpa() method, 639  
gpas collection, 759  
gpaTextBox text box, 757  
gpaTextBox\_TextChanged()  
method, 758  
gradePointAverage column, 734  
GradePointAverage field, 387, 738, 743, 766  
GradePointAverage property, 388–389  
graphical user interfaces (GUIs), **6**, 39, 104  
widgets, **558**  
greater than (>) operator, 71, 736  
greater than or equal to (>=) operator, 71, 736  
Greenwich Mean Time (GMT), 676  
Greeting class, 629  
Greeting2 class, 631  
Greeting program, 629–630  
Greeting2 program, 631  
greeting variable, 458  
GreetingDelegate() delegate, 629  
greetingsButton\_Click() method, 227–228

GreetMethod() method, 629  
gross parameter, 357  
grossPay variable, 302  
group clause, 768  
group operator, **766**–768, 776  
GroupBox class, 582, **600**, 617  
grouping  
data, 765–773  
records, 766–768  
Grow() method, 442  
GUI applications, 104, 127  
arrays, 271–273  
automatically generating  
methods, 315–316  
correcting errors, 123–126  
decision making, 180–182  
enhancements, 127  
enumeration, 183–184  
event driven, **624**  
formatted strings, 120  
inheritance, 487–489  
logical errors, 125  
loops, 227–230  
methods, 315–316  
objects, 422–424  
placing controls on forms, 182  
scope, 316  
switch structure, 183–184  
syntax errors, 123–124  
text file creation, 708–712  
GUIs. *See* graphical user interfaces (GUIs)

**H**

hardware, **2**, 38  
Harvard University, 3  
has-a relationship, **375**, 427  
hash codes, **469**–470, 492  
HeartSurgeon class, 445  
heatingBill variable, 48  
Height property, 405, 560–562, 600  
Hello application, 33  
Hello class, 21  
Hello folder, 35  
Hello() method, 629, 631  
Hello namespace, 33  
HelloClass program, 284–285, 290–292  
Hello.cs file, **21**–**22**, 31, 35  
Hello.exe file, 31–32, 35  
HelloForm form, 130  
helloLabel label, 132, 603, 624  
HelloVisualWorld program, 129, 132  
help for Visual Studio, 126  
hexadecimal numbering system, **793**, 796  
hexadecimal shorthand, 77–78, 95

HIGH constant, 153  
high-level programming languages, **2**–**3**, 38  
highNums collection, 744  
HIGHPRICE constant, 763  
HonestRalphsUsedCars database, 751–754  
HonestRalphsUsedCars.accdb file, 770  
horizontal tab (**\t**) escape sequence, 77  
hot keys, 582  
hours parameter, 357  
hoursWorked variable, 72

**I**

IAttackable interface, 483  
IComparable interface, **414**–**415**, 419, 428  
ID field, 751  
IDE. *See* Integrated Development Environment (IDE)  
identifiers, **4**–**5**, 39, 387  
beginning with underscore, at sign (@), or letter, 14  
Boolean variables, 71  
consecutive values, 84  
contextual keywords, **381**  
keywords, 14  
methods, 289–290  
objects, 375–376  
properties, 381  
representing constants, 84–86  
selecting, 14–16  
special meaning, 15  
style, 5  
idNumber field, 373–374, 378–379, 381–382, 389, 415, 421, 734  
idNumber primary key, 734  
IdNumber property, 381–382, 384–385, 388–389, 401, 404, 413, 419  
idNumbers array, 261  
idNumbers integer, 260  
if-else statements, **155**–**159**, 180  
abbreviated version, 173–174  
blocks, 155  
indentation, 155  
nested, 156–157, 178  
if keyword, 155  
if statements, **147**–**154**, 175–176, 180, 186, 251–252, 511, 540, 608, 610, 635  
blocks, 150–151  
Boolean variables, 147  
compound expressions, 160–165, 167  
conditional OR operator (||), 162  
else clause, 177

- equivalency comparisons, 153–154  
 evaluation order, 223  
 false expressions, 147, 149  
 incorrectly inserting semicolon, 149  
 indentation, 149  
 multiple statements and, 150  
 nested, 151–152, 161, 168, 171  
 parentheses, 168  
 range checks, 177–178  
 true value, 147
- IL. *See* intermediate language (IL)
- Image property, 585–586, 609  
 images, selecting, 586  
 immutable, 82, 95  
 implementation hiding, 285, 297, 318, 356  
 implicit cast, 74, 94  
 implicit conversions, 74–75, 94, 462, 492  
 implicit numeric conversions, 74–75  
 implicit parameters, 382, 427  
 implicit reference conversions, 462–463, 492  
 implicitly, 74, 94  
 implicitly typed variables, 740–742, 776  
 #include files, 10  
 incrementing, 202, 231  
 indefinite loops, 202, 206, 231  
 index, 242, 274  
 IndexOutOfRangeException  
     exception, 247, 521–524, 532–533, 543  
 inexpensiveCars collection, 763–765  
 inexpensiveCarsBox list box, 760  
 inFile file, 697  
 infinite loops, 198–199, 201, 206, 231, 461  
 Infinity value, 506  
 information hiding, 373–374, 427, 448–450  
 inheritance, 8, 39, 438–442, 491  
     base class, 441  
     benefits, 440, 488–489  
     classes, 475  
     default constructors, 470  
     derived classes, 441, 449  
     extended classes, 441  
     GUI applications, 487–489  
     interfaces, 482  
     multiple, 10  
     one direction, 445  
     transitive, 442  
     Windows Forms applications, 487  
         working example, 446–448  
 initialization, 50–51, 93  
 InitializeComponent() method, 424, 564–565, 579, 589  
 initializer list, 245, 275  
 initializing arrays, 245–246  
 inner loop, 216, 218, 231  
 input stream, 683  
 Input string was not in a correct format exception, 334  
 InputMethod() method, 332  
 InputMethodDemo program, 332  
 instance methods, 81, 371, 415, 426  
     explicitly referring to this reference, 394  
     public, 374  
 instance variables, 371, 373–375, 393, 426, 704  
 instances, 7, 39, 370, 426  
     destroying, 420–422  
     out of scope, 420, 422  
 instantiate, 370, 426  
 instantiating objects, 370, 372–375  
 instantiation, 370, 426  
 int alias, 464  
 Int32 class, 484–485, 489  
 int data type, 48, 51, 57–58, 60, 93, 169  
 int variable, 86  
 Integer class, 527  
 integer constants, 65  
 integer return type, 314  
 integer variables, 65, 69  
 integers, 90, 93, 732  
     addition, 407  
     converting input to, 531  
     divide by zero error, 506, 521  
     division, 65  
     negative, 58  
     positive, 58  
     remainder (modulus) operator, 65  
 integral data types, 57–58, 93  
 Integrated Development Environment (IDE), 22–23, 41, 381  
     adding functionality to buttons, 112–115  
     automatically generated code, 563–566  
     C++, 104  
     changing properties of multiple objects, 568  
     Designer.cs file, 626  
     double-clicking LinkLabel object, 571  
     drag-and-drop design features, 566  
     Error list, 105  
     examining code generated by, 579–581  
 executing programs, 118  
 FILE menu, 105  
 form creation, 104–109  
 Form Designer, 105  
 main window, 105  
 method shell, 571  
 project name, 105  
 Properties list, 575  
 Properties window, 105  
 Solution Explorer, 105  
 switching views, 112  
 Toolbox tab, 105, 109  
 VIEW menu, 106  
 IntelliSense, 797–799, 800  
     completion mode, 797  
     information about methods, 346  
     listing defined values for enumeration, 85  
     suggestion mode, 798  
     using statement, 799  
 interactive programs, 86–91, 95  
 InteractiveAddition file, 90–91  
 InteractiveSalesTax program, 87, 89  
 interestRate variable, 67  
 interface keyword, 480  
 interfaces, 8, 39, 285, 318, 413, 428, 479, 492  
     abstract methods, 479–483  
     classes implementing, 414  
     deciding which to use, 127–128  
     defining named behaviors, 414  
     inheritance, 482  
     methods, 414  
     names, 480  
     polymorphism, 414, 483  
 intermediate language (IL), 22, 41  
 internal class access modifier, 372, 427  
 internal field modifier, 373  
 internal keyword, 372  
 interpreters, 3  
 intrinsic types, 48–49, 92  
 InvalidPrinterException class, 504  
 invoked, 284, 318  
 invoked methods, 318  
 invoking  
     events, 639, 645, 666  
     methods, 5, 12, 39  
     objects, 377, 427  
 IOException class, 504, 529, 686  
 IPlayable interface, 482  
 ISerializable interface, 704  
 isLocalCustomer() method, 304  
 isPreferredCustomer() method, 304  
 isSixMore variable, 73  
 IssueInvitations() method, 371  
 isValidIDNumber variable, 147

isValidItem Boolean variable, 252–253  
 isValidItem variable, 256  
 itemOrdered variable, 245, 252–254, 255  
 itemPrice variable, 253–255  
 itemPriceAsString variable, 87–88  
 Items property, 588–589  
 Items.Count property, 591  
 iteration, **198**, 202, 231  
 iteration variables, **248**, 275, **741**, 776  
 IWorkable interface, 480–483

**J**

jagged arrays, **270**–271, 275, 311  
 Java, 2–3, 5  
     ban on passing functions, 628  
     based on C++, 10  
     built-in methods, 10  
     declaring array, 242  
     finally block, 530  
     shorter alias, 50  
     simple data types, 10  
     try and catch blocks, 530  
 just in time (JIT) compiler, **22**, 41

**K**

K & R style, **151**, 186  
 Kaufman, George, 558  
 KB or kB. *See* kilobytes (KB or kB)  
 Kernighan, Brian, 151  
 key events, **657**–659, 666  
 key field, **682**, 720  
 Key value, 766  
 keyboard  
     events, 657–659  
     input, 683  
 KeyDemo program, 659  
 KeyEventArgs class, 657  
 KeyPressEventArgs class, 657  
 keys, **729**, 733, 775  
 KeyUp event, 658  
 KeyUp() method, 658  
 keywords, **2**, 13, 38, 50  
     identifiers, 14  
     Language INtegrated Query (LINQ), 744  
     reserved, 14–15  
     switch structure, 169  
 kilobytes (KB), **675**, 720, 794–795

**L**

Label class, 115, 563, 565, 583, 586, 598, 603, 611, 650–651, 653, 660–661, 709, 712  
 label1 control, 577, 598  
 label2 control, 598

label1 identifier, 563  
 labels, **115**–**121**, 132–133, 136, 558, 575–582, 598, 709  
     blank, 116  
     changing fonts, 120–121  
     changing properties, 575–581  
     currency, 120  
     explanatory text for value, 120  
     invisible, 133  
     linking user to other sources, 569–572  
     multiple lines of text, 115–116  
     resizing, 115  
     text, 115, 603  
 Labels control, 117  
 Language INtegrated Query (LINQ), **743**–**746**, 776  
     complicated sequences, 745  
     keywords, 744  
     operators, 744  
     operators to sort and group data, 765–768  
     AND and OR expressions, 758  
     queries, 770  
     queries with Access database  
         table, 755–759  
         query syntax, 743–744  
 largeToolStripMenuItem\_Click() method, 603  
 lastName column, 734, 736  
 lastName field, 389, 728, 738, 756  
 LastName property, 388–389  
 LeavesTurnColor() method, 442  
 left-associative operators, **785**, 786  
 Length property, **82**, 95, 247–**248**, 259–266, 275, 340, 391–392, 684  
 less than (<) operator, 71, 736  
 less than or equal to (<=) operator, 71, 736  
 Leszynski naming convention (LNC), **731**, 775  
 letterButton\_Click() method, 661  
 lexically, **81**, 95  
 LIMIT constant, 199–200, 207–208  
 LincolnCheckBox class, 605  
 lincolnCheckBox\_CheckedChanged() method, 610  
 LincolnForm form, 609  
 LincolnRoomCheckBox class, 608, 610, 614  
 line comments (//), **18**, 40  
 LinkClicked() method, 570–572, 650  
 LinkColor property, 570  
 LinkLabel class, **569**–**572**, 570, 616, 650  
 linkLabel1 identifier, 571

linkLabel2 identifier, 571  
 links, 569–572  
 LinkVisited property, 570, 572  
 LINQ. *See* Language INtegrated Query (LINQ)  
 LINQ queries, 760–762, 771–772  
 LINQ statements, 407, 741–742  
 LinqDemo1 program, 744  
 LinqDemo2 program, 745–746  
 Linux, 2  
 list boxes, 589–592, 764  
 ListBox class, 588, **589**, 617  
 ListBox objects, 588–592, 755–756, 758, 766  
 ListControl objects, 588–592  
 lists, 588–592  
 literal characters, 76  
 literal constants, **48**, 92  
 literal strings, **11**, 40  
 LNC. *See* Leszynski naming convention (LNC)  
 Load() method, 590  
 Loan class, 446, 459, 461, 474  
 Loan demonstration program, 446–448  
 Loan objects, 446, 448, 460  
 loanAmount field, 460, 461  
 LoanAmount property, 460  
 LoanNumber property, 461  
 local variables, **292**, 319  
 LocalVariableDemo program, 292  
 Location object, 587  
 Location property, 131–132, 424, 565, 578, 597, 626, 752  
 logic, **3**, 38  
 logic-planning tools, 144–147  
 logical AND operator, 251  
 logical errors, 3  
     GUI applications, 125  
 logical operators, 163  
 logical OR operator (||), **568**, 616  
 Logo, 4  
 long data type, **57**, 60, 84, 93, 169  
 loop body, **198**, 201, 212–213, 231  
 loop control variable, **199**, 231  
     altering value, 199  
     comparing to zero, 223–224  
     decrementing, **202**  
     incrementing, **202**, 225  
     initializing, 207  
     starting value, 206  
     step value, **207**  
     testing, 199, 207  
     updating, 207  
 loop fusion, **225**, 231  
 LoopingBankBal program, 203, 206  
 LoopingBankBal2 program, 216–217



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

NullReferenceException exception, 717  
 number alias, 331–332  
 number variable, 153, 199–201, 526–527  
 numbering systems  
   binary, 788–792, 795  
   decimal, 787, 795  
   hexadecimal, 793, 796  
 numberOfDependents variable, 58  
 numbers, 57–58  
   as character, 76  
   converting into string, 63  
   converting string to, 89, 333–336  
   converting value to, 87  
   currency values, 63–64  
   floating-point, 61–64  
   formatted string, 64  
   significant digits, 61  
 numbers array, 741, 744  
 numeric constants, 568  
 numeric data type  
   arithmetic, 76  
   tables, 732  
 numeric fields and array elements, 245  
 numeric type conversion, 74–75  
 numeric variables, assigning value, 58  
 numOfItems variable, 257–259  
 numPages field, 393

**O**

Oak class, 442  
 obj folder, 35  
 Object argument, 464  
 Object class, 337, 414, 464, 488, 492, 507, 558  
   Equals() method, 467–468  
   GetHashCode() method, 469–470  
   GetType() method, 466  
   methods, 465  
   public instance methods, 465–466  
   ToString() method, 466–467  
 Object constructor, 470  
 object destructors, 10  
 object initializers, 404–407, 428  
 object keyword, 464  
 object-oriented approach, 6, 39  
 object-oriented exception-handling methods, 512–517  
 object-oriented programming languages, 5, 7–9  
 object-oriented programming (OOP), 5–6, 39

object-oriented programs, 8  
 Object parameter, 468  
 objects, 5–7, 10, 39, 370, 464–470, 492  
   address of invoking, 394–395  
   aliases, 464  
   within another object, 375  
   arithmetic symbols, 408  
   attributes, 5, 7–8  
   behaviors, 5–6  
   belonging to specific class, 7–8  
   changing properties for multiple, 568  
   classes, 466  
   cleanup, 558  
   comparing, 414, 467–468  
   containment, 558  
   converting into streams of bytes, 704–707  
   creation, 375–379, 386–389  
   data types, 375–376  
   declaring and reserving memory, 376  
   default value of, 397  
   defining, 375–376  
   different starting values, 405  
   disposal, 558  
   fingerprint, 469  
   generating event by clicking, 570  
   GUI applications, 422–424  
   hash codes, 469–470  
   identifiers, 375–376  
   instance variables, 371, 392–396  
   instantiating, 370, 372–375  
   interface with methods, 8  
   invoking, 377  
   manipulating reference to, 558  
   methods, 8, 371, 396–403  
   names, 376, 412, 582  
   passing to methods, 378–379  
   properties, 7  
   receiving attributes from classes, 371  
   reference type objects referring to same, 467–468  
   state, 5, 371  
   throwing, 543  
   types, 466  
    uniquely identifying, 469–470  
 Obstetrician class, 445  
 offending method, 535  
 okButton button, 114, 315, 563  
 okButton\_Click() method, 272, 315–316  
 OleDbCommand type, 743  
 OnBalanceAdjusted event handler, 645  
 OnBalanceAdjusted() method, 645  
 OnChanged() method, 639  
 one-dimensional arrays, 266–267, 275  
 One value, 589–590  
 OneButtonTwoEvents project, 651  
 oneVal parameter, 309  
 oneVal variable, 309  
 OOP. *See* object-oriented programming (OOP)  
 Open mode, 685, 695  
 opening files, 682, 720  
 OpenOrCreate mode, 685  
 operands, 64, 94, 173  
 Operate() method, 453  
 operating system error messages, 23–24  
 operator -() method, 411  
 operator \*() method, 411  
 operator +() method, 409, 410  
 operator precedence, 94  
   arithmetic operators, 65–66  
 operators  
   aggregate, 759  
   associativity, 66, 785, 786  
   binary, 408  
   grouping data, 765–768  
   identifier, 408  
   Language INtegrated Query (LINQ), 744  
   order of operation, 66  
   overloaded, 340, 408  
   overloading, 407–411  
   precedence, 165, 785–786  
   projection, 744  
   restriction, 744  
   sorting data, 765–768  
   unary, 408  
 optional parameters, 328, 351–359, 361  
 OptionalParameterDemo program, 352  
 OR operator, 162, 186, 736  
 Oracle XE, 738  
 order of operation, 66, 94  
 orderby operator, 765–766, 771, 776  
 orderDay variable, 85  
 orphaned methods, 125, 136  
 out arguments, 418  
 out keyword, 330, 334  
 out modifier, 328  
 out of scope, 208, 231  
 out parameter, 334–335  
 outer loop, 216, 218, 231  
 outFile stream, 685, 705  
 output, 11–13  
   displaying on screen, 11–12, 683  
   ending with carriage return, 688

output parameters, 328, **330**, 332–333, 360  
 output stream, 683  
 outputLabel label, 183, 227, 583  
 outputListBox list box, 755–756  
 overload resolution, **345**, 359, 361  
 overloaded constructors, 402–403, 568  
 overloaded methods  
     ambiguous, 350  
     betterness rules, **345**–346  
     built-in, 346  
     data types, 344  
     inability to choose, 349–351  
     optional parameters, 355  
     overload resolution, **345**  
     parameter identifiers, 344  
     return types, 344  
     reusing, 344  
 overloaded operators, 340  
 OverloadedTriples class, 348  
 OverloadedTriples program, 348–349  
 overloading, 467  
     constructors, 398–400  
     methods, **340**–348, 360  
     operator `*`() method, 411  
     operators, 407–411  
     symbols, 408  
     unary operators, 411  
 override keyword, 455, 464, **476**–477, 492  
 overrides, **414**, 428  
 overriding  
     abstract methods, 476  
     base class members, 453–458  
     `Equals()` method, 467–468  
     `GetHashCode()` method, 469  
     `ToString()` method, 466

**P**

Painter class, 475  
 Panel control, 582, **600**, 617  
 parallel arrays, 252–253, 258, 275  
 parameter arrays, 328, **330**, 336–337, 360  
 parameter identifiers and overloaded methods, 344  
 Parameter Info ToolTip, 799  
 parameter lists, **290**, 319, 344  
     constructors, 398  
     default variable value, 353  
     optional parameters, 351–359  
 ParameterDemo2 program, 331  
 parameterless constructors, **398**–399, 404, 428

parameters  
     default values, 353  
     implicit, **382**  
     local identifier, 296  
     mandatory, **328**  
     methods, **289**–290, 296, 319  
     multidimensional arrays, 311  
     optional, 328, 351–359  
     output, 328, **330**, 332–333  
     passing to constructors, 397–398  
     reference, 328, **330**, 333  
     types, 328–329  
     value, 328, 333  
 params keyword, 330, 336  
 params modifier, 328  
 ParamsDemo program, 337  
 Parent class, 482  
 parent classes, **441**–442, 491  
     general, 475  
     hides counterpart, **455**  
     private field, 450  
     public field, 450  
     public set accessor, 450  
     subclasses inheriting from multiple, 479  
 parse, **89**, 95  
 Parse() method, 89, 333  
 parsing strings, 89  
 partial contextual keyword, 381  
 partial identifier, 15  
 Party class, 371  
 Pascal casing, **5**, 39  
 PassArrayElement program, 308–309  
 passed by reference, **309**, 319  
 PassEntireArray program, 310–311  
 Patent class, 479  
 path command, 24  
 paths, **675**, 709, 720  
     changing default, 24  
     command prompt, 31  
 PathTooLongException class, 504  
 payRate array, 248–249, 741  
 PayRate constructor, 398  
 payRate variable, 67, 178  
 Payroll program, 483  
 PayrollApplication class, 289  
 PB. *See* petabytes (PB)  
 pepperoniCheckBox object, 582  
 PerformLayout() method, 564  
 permanent storage devices, **674**, 719  
 persistent storage, **675**, 720  
 petabytes (PB), 795  
 PhoneCall class, 306  
 picture boxes, 585–587  
 PictureBox control, 609, 616  
 pictureBox1 object, 587

PictureBox objects, **585**–587  
 PizzaOrder program, 584  
 pizzaPrice value, 582  
 placeholders, **53**–54, 93  
 Play() method, 453  
 Pointer tool, 607  
 pointer variable, 628  
 polymorphic behavior, 483  
 polymorphism, **9**, 39, 414, 453  
 Poodle class, 441  
 Position property, 684, 706  
 positional arguments, **354**, 361  
 positive integers, 58  
 positive value, 407  
 postfix decrement operator `(--)`, **68**  
 postfix increment operator `(++)`, **67**–68, 94  
 postfix incrementing loops, 225–226  
 posttest loops, **215**, 231  
 Pow() method, 305  
 precedence, **785**–786  
 precision specifier, **62**–63, 93  
 predetermined iterations, 202  
 PreferredDiscount() method, 633, 637  
 prefix decrement operator `(--)`, 68  
 prefix increment operator `(++)`, **67**–68, 94  
 prefix incrementing loops, 225–226  
 PrefixPostfixComparison program, 226  
 pretest loops, **215**, 231  
 Preview Changes dialog box, 122  
 price field, 393, 751  
 priceLabel label, 611–613  
 PriceList class, 532–533, 537  
 PriceListApplication1 program, 532–533  
 PriceListApplication2 program, 533–534  
 priceOfCall variable, 306–307  
 prices array, 245, 252  
 priceTextBox text box, 762  
 primary keys, **729**, 733–734, 775  
 private access, **287**–288, 319  
 private access modifiers, 389–392  
 private access specifier, 114, 315  
 private class access modifier, **372**, 427  
 private data fields, 389, 448  
 private fields, 380, 383, 390, 449  
     modifier, 373  
     parent class, 450  
 private keyword, 295  
 private methods, 390  
 private modifier, 287–288  
 private static, 290–292

procedural programming languages, 4  
 procedural programs, 4–5, 38  
 procedures, 5  
 Product class, 479  
 product variable, 69  
 Program class, 28, 33  
 program comments, 17–18, 36–37, 40  
 Program.cs file, 106  
 programming, 2–3  
 programming languages  
     error messages, 23, 25, 27  
     high-level, 2–3  
     semi-compiled, 22  
     syntax, 3  
     syntax errors, 3  
     variables, 4  
     vocabulary, 3  
 programs, 2, 38  
     allowing user input, 86–91  
     arithmetic statements, 69–70  
     comments, 36–37  
     compiling, 22–37  
     debugging, 3  
     deciding which environment to use, 29–30  
     design time, 118  
     differences between text editors  
         and Visual Studio IDE, 28  
     documenting, 17–18  
     entering, 21–22  
     event-driven, 181  
     exceptions, 504–507  
     executing, 22–37  
     fault-tolerant, 507  
     indentation, 12–13  
     interactive, 86–91  
     interface, 104  
     logic, 3  
     main logic interrupted, 511  
     malfunctioning, 18  
     methods, 5  
     multifile assembly, 285  
     planning logic, 144  
     procedural, 4–5  
     robustness, 507  
     run time, 118  
     self-documenting, 356  
     statements, 19  
     translating into machine language, 3  
     whitespace, 12–13  
 projection operators, 744, 776  
 projects, 33  
     adding dataset, 751–754  
     closing, 132  
     data source, 747  
     earliest planning stages, 6

executing, 132  
 files part of current, 106  
 names, 105  
 object-oriented approach, 6  
 opening, 129  
 prompt, 95  
 propagating exceptions, 536, 548  
 properties, 7, 39, 380, 427  
     access modifiers, 381  
     accessors, 380  
     assigning values to, 404  
     auto-implemented, 383–385, 384, 389  
     automatic, 384  
     backing fields, 381  
     base class, 457–458  
     creation, 379–386  
     data types, 381  
     declaration, 381  
     description, 107  
     expanding or condensing, 108  
     explicitly referring to this reference, 394  
     get accessors, 380, 381  
     identifiers, 381, 387  
     names, 108  
     organizing, 107  
     public, 448  
     read-only, 381, 392, 562  
     set accessors, 380–381  
     as simple variable, 382  
     writing to, 381  
 Properties folder, 35  
 Properties list, 107, 576, 578, 581  
 Properties window, 105, 107–108, 129–130, 136, 576, 598  
     Events icon, 651  
     listing events, 652  
     project name, 105  
     Properties icon, 652  
     property names, 108  
 property identifiers, 248  
 proportional font, 120, 136  
 protected access specifier, 448–452  
 protected class access modifier, 372, 426  
 protected field modifier, 373  
 protected fields, 450  
 protected keyword, 492  
 protected methods, 450  
 protected properties, 560  
 pseudocode, 144, 185  
 public access, 287–288, 314, 319  
 public access modifiers, 389–392  
 public access specifier, 315  
 public class access modifier, 372, 426  
 public data fields, 391

public field modifier, 373  
 public fields, 390, 450  
 public instance methods, 374  
 public methods, 380, 389–390, 409, 448  
 public modifier, 287–288  
 public properties, 448, 461, 560  
 public set accessor, 450  
 purchase variable, 221  
 PushButtonControl class, 15, 16

## Q

queries, 735, 743–744, 768, 775  
 query by example, 735, 775  
 querying datasets, 760–762  
 Quick Info ToolTip, 798  
 Quicksort algorithm, 262  
 quotient variable, 69

## R

radio buttons, 581–584  
 RadioButton class, 559, 581–584, 611–614, 616  
 RadioButton objects, 581–582, 584  
 raises an event, 624, 666  
 random access files, 682, 720  
 random access memory (RAM), 674, 719  
 range checks, 177–178, 187  
 range match, 257, 258, 259, 275  
 RATE constant, 454  
 rate parameter, 357  
 rate variable, 67  
 rateLabel property, 578  
 Read access, 685, 695  
 Read() method, 683  
 read-only properties, 248, 381, 385, 392, 427  
 readability aids, 800  
 reading  
     data from text file into forms, 712–718  
     from files, 675, 684, 695–696, 720  
     from sequential access files, 690–692  
     from text files, 703–707  
 ReadLine() method, 88, 127, 261, 284, 288, 302, 333, 683, 690–691  
 ReadNameFile file, 695–696  
 readonly field, 402  
 readonly field modifier, 373  
 readonly keyword, 252  
 readonly modifier, 402  
 ReadSequentialFile program, 691–692, 697

ReadWrite access, 684–685  
 record delimiter, 688  
 recordIn string, 690  
 records, **681**, 720, 728–729  
     containing many fields, 703  
     grouping, 766–768  
     primary keys, **729**  
     selecting all fields, 736  
     sorting, 682, 729  
 recordsButton\_Click() method, 754, 761, 763–764  
 rectangular arrays, **267**, 275  
 recursive methods, **462**, 492  
 Red button, 663–664  
 redButton\_Click() method, 664  
 ref keyword, 330  
 ref modifier, 328, 331  
 refactoring code, 122  
 reference equality, **467**–468, 492  
 reference parameters, 328, **330**, 333, 338–339, 360  
 reference types, **376**, 412, 427  
 relational databases, **730**, 743, 775  
 relationships and tables, 734  
 Release folder, 35  
 remainder ((modulus) %) operator, 65  
 remainder variable, 69  
 remove identifier, 15  
 Rename dialog box, 122  
 RentFinder program, 269  
 rents array, 270  
 RepairValve() method, 445  
 reserved keywords, 14–15  
 resources, 106  
 response variable, 204  
 restriction operators, **744**, 776  
 result variable, 305  
 ResumeLayout() method, 564  
 ReThrowDemo program, 545–546  
 rethrowing exceptions, **544**–546, 548  
 return statements, 169, 289, **302**, 319, 529  
 return types, **289**, 319, 344, 381  
 Reverse() method, **262**–266, 275, 309  
 ReverseArray program, 263  
 right-associative operator, **785**  
 Riley, James Whitcomb, 740  
 Ritchie, Dennis, 151  
 robustness programs, **507**, 548  
 root class, 492  
 root directory, **675**, 720  
 runnable, **13**, 40  
 runtime, **118**, 136, **624**, 666  
 runtime errors, 504

**S**  
 sal parameter, 401  
 salary field, 397  
 Salary property, 397, 401, 404  
 saleAmount parameter, 300  
 saleAmount variable, 177–178, 296–297  
 sales array, 267  
 salesAmountForYear variable, 163  
 salesCode variable, 179  
 salesperson object, 443  
 SatisfyGraduationRequirements() method, 453  
 sausageCheckBox object, 582–583  
 sausageCheckBox\_CheckedChanged() method, 583  
 Save the Connection String dialog box, 753  
 saving memory, 58  
 sbyte data type, **57**–**58**, 60, 84, 93, 169  
 ScholarshipStudent class, 455, 464  
 scientific notation, 93  
 scope, **292**, 319  
 score variable, 89, 333, 336  
 scores array, 264–265  
 screen output, 683  
 ScrollableControl class, 488  
 Sculptor class, 475  
 sealed classes, **484**, 493  
 searching arrays  
     loops, 251–259  
     for loops, 252–254  
     objects, 413–417  
     range match, 257–259  
     while loop, 255–256  
 searching sequential text file, 697–699  
 secondDel delegate, 629, 631, 633, 635  
 Seek() method, 694, 697, 699–703  
 seekEmp object, 417  
 SeekOrigin enumeration, 697  
 select clause, 768  
 SELECT command, 743  
 SELECT-FROM-WHERE SQL command, **735**–**736**, 775  
 select keyword, **744**, 776  
 Select Resource window, 586  
 SELECT SQL statements, 736–737  
 SelectedIndex property, 588  
 SelectedIndexChanged() method, 588  
 SelectedIndices property, 588  
 SelectedItem property, 588–589, 591  
 SelectedItems property, 588, 592

SelectionEnd property, 594  
 SelectionMode property, 588–590  
 SelectionRange property, 594  
 SelectionStart property, 594  
 self-documenting, 95, 361  
 self-documenting programs, **356**  
 self-documenting statements, **84**  
 semantic errors, **3**, 38  
 semi-compiled programming languages, 22  
 sender object, 625  
 sentinel value, **203**, 231, 689  
 sequence structure, **144**, 185  
 sequential access files, **681**–682, 720  
     key field, **682**  
     reading from, 690–692  
     writing data to, 687–690  
 sequential search, **252**, 275  
 sequential text file, searching, 697–699  
 serializable, 704  
 [Serializable] attribute, 704  
 SerializableDemonstration program, 705–707  
 serialization, **704**–707, 721  
 Serialize() method, 705  
 set accessors, **380**–**384**, 387, 389, 427, 443, 449–450, 453–454, 460–461, 541, 543, 639  
 set contextual keyword, 381  
 set identifier, 15  
 SetIdNumber() method, 379  
 SetPriceAndTax() method, 395  
 SetSelected() method, 592  
 setter, **381**, 427  
 settings, 105  
 short-circuit evaluation, **161**–**163**, 186, 304, 358  
 short-circuit operators, 222–223  
 short data type, **57**–**58**, 60, 84, 93, 169  
 shortcut arithmetic operators, 66–68  
 shortcut operator, 200  
 shorter alias, 50  
 showButton button, 756  
 showButton\_Click() method, 755–758  
 ShowDialog() method, 608  
 ShowToday property, 594  
 ShowTodayCircle property, 594  
 ShowWelcomeMessage() method, 290–292  
 side effects, **163**, 186, 304, 358  
 signature, **340**, 360  
 significant digits, **61**, 93  
 simple data types, **48**–**49**, 92, 704  
 SimpleMethod() method, 349–350

- single-dimensional arrays, **266**, 275  
 single quotation mark (') escape sequence, 77  
 Size property, 107–108, 112, 115, 129, 424, 565, 587, 624, 626  
 SizeMode property, 585  
 Smart Tag, 799  
 snap lines, **597**, 617  
 SoapException class, 504  
 socialSecurityNumber field, 728  
 software, **2**, 38  
 Solution Explorer, **105**–107, 136, 580, 748  
 someMoney variable, 52–55  
 someNums array, 308–309  
 SomeOtherClass class, 292  
 someWorker object, 466  
 Sort() method, **261**–262, 264–266, 275, 309  
     array of objects, 413–417  
 SortArray program, 262  
 Sorted property, 588  
 sorting  
     array items, 265  
     array of strings, 262  
     arrays of objects, 413–417  
     ascending, 766  
     compound conditions, 766  
     data, 765–768  
     descending, 766  
     records, 682, 729  
 source code, **22**, 25, 41  
 Speak() method, 476–478  
 special symbols, 78  
 specialButton class, 183–184  
 specialButton\_Click() method, 184  
 Split() method, 691  
 spreadsheets, 267  
 Spruce class, 441–442  
 SQL. *See* Structured Query Language (SQL)  
 StackTrace() method, 541  
 StackTrace property, 536–540, 543  
 standard error stream object, 683  
 standard input stream object, 683  
 standard numeric format strings, **62**, 93  
 standard output stream object, 683  
 StandardDiscount() method, 632–633, 637  
 Start() method, 572  
 Start Without Debugging command, 27  
 StartsWith() method, **83**, 95, 758  
 state, **5**, 39, **371**, 426  
 statements, 19  
     blocking single, 151  
     chaining addition in, 411  
     commenting out, **18**  
     declaring multiple variables of same type, 51  
     ending with semicolon, 11  
     multiple-line, 51  
     self-documenting, **84**  
 static, 291  
 static classes, 489  
 static field modifier, 373  
 static fields, 391  
 static keyword, **13**, 40, 114, 252, 295, 316, 374, 391  
 static keyword modifier, 288–289  
 static methods, 81, 114, **288**–289, 295, 297, 306, 319, 374, 409, 485, 637  
 static modifier, 287, 374  
 static named constants, 391  
 STEP keyword, 207  
 step value, **207**, 231  
 stopping loops, 203  
 Stopwatch class, 223–224  
 storage  
     measuring, 794–795  
     nonvolatile, **674**  
     permanent, 674  
     persistent, **675**  
     temporary, 674, 680  
     volatile, **674**  
 storage devices, 675  
 StreamReader class, 684, 690, 695–696, 702, 713  
 streams, **683**–687, 720  
 StreamWriter class, 684, 685, 688, 693–694, 710  
 StreamWriter constructor, 685  
 string[] args parameter, 313  
 string argument, 568, 629  
 String class, 340–342, 413, 489, 691  
 String Collection Editor, 589  
 string data type, 48, 79–83, 95, 169  
 string literal, 76  
 string objects, 378–379, 468  
 string return type, 302  
 string variables, 80, 86  
 String.Format() method, 54, 120  
 strings, 79–83, 713  
     @ sign, 710  
     arguments, 11  
     array of characters, 216  
     assigning user input to, 86–91  
     characters, 76  
     collecting from table, 757  
     comparing, 80–82  
     concatenating, 53, 56–57, 407  
     converting, 63, 87, 89, 333–336, 526–527  
     displaying, 53  
     escape characters, 120  
     format string, **53**–**54**  
     formatted, 64, 120  
     formatting, 54–56  
     immutable, 82  
     initializing with character array, 80  
     length, 82  
     name of class, 466  
     parsing, 89  
     placeholders, **53**–**54**  
     separating data fields into array of, 691  
     sorting array of, 262  
     substrings, 82  
 strongly typed, **742**, 776  
 Structured Query Language (SQL), **735**–**737**, 775  
 stu object, 639  
 Student class, 386–389, 402, 453–456, 464, 638–643, 681, 745  
 Student constructor, 418  
 Student objects, 386, 388, 418, 589, 639–640  
 StudentChanged delegate, 639  
 StudentChanged() method, 640  
 sub variable, 246  
 subclass-superclass relationship, 443  
 subclasses, **441**, 491  
     ancestors, **442**  
     inheriting from more than one parent class, 479  
 subroutines, 5  
 subscripts, **242**, 244, 246–247, 274  
 Substring() method, **82**, 95  
 substrings, 82  
 Subtraction (-) operator, 65  
 SUDSParserException class, 504  
 suggestion mode, **798**, 800  
 Sum() aggregate operator, 759  
 sum variable, 69, 90  
 sumButton button, 122  
 Sun StarBase, 738  
 sunny day case, 511  
 superclass, **441**, 475, 491  
 SuspendLayout() method, 564  
 Swap() method, 338  
 swapping values, 338–339  
 SwapProgram program, 338–339  
 switch expression, **169**, 186  
 switch keyword, **169**, 186  
 switch statements, 168–172, 180  
 switch structure, **168**–**172**, 186  
     GUI applications, 183–184  
 symbolic logic, 71  
 symbols, overloading, 408



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Visual Studio (*continued*)  
 Integrated Development Environment (IDE), 22  
 IntelliSense, 346  
 locating line of code, 564  
 namespaces, 12  
 New Project window, 576  
 nodes, **106**  
 projects, 33  
 using another programming language in, 30  
 Visual Studio IDE, 104, 128–134  
 advantages, 26  
 automatic statement completion, 29  
 built-in tools, 747–750  
 compiling code, 26–28, 32–35  
 correcting errors, 30  
 differences between programs, 28  
 executing programs, 32–35  
 Help facility, 662  
 main menu strip, 601  
 programming language error message, 27  
 selecting from list of choices, 29  
 words displayed in different colors, 30  
 Visual Studio Web site, 126  
 void, 290–292  
 Void class, 16  
 void keyword, **13**, 40  
 void methods, 296  
 void return type, 289–292, 301, 312, 629–630, 630  
 volatile, **674**, 719  
 volatile field modifier, 373

**W**

warnings, 25  
 wasSuccessful variable, 527  
 wavy lines in code, 799  
 weeklyBudget variable, 75

WelcomeMessage() method, 374, 377, 381  
 where clause, 756  
 where identifier, 15  
 where keyword, 381, **744**, 776  
 while expression, 259  
 while keyword, 198  
 while loops, **198**, 213, 215, 227, 231, 246, 533, 702  
 checking value at “top” of loop, 212  
 curly braces, 201  
 definite loops, 206  
 indefinite loops, 206  
 loop control variable, **199**  
 searching arrays, 255–256  
 White button, 664  
 whiteButton\_Click() method, 664  
 whitespace, **12**–13, 40  
 widgets, **558**, 616  
 Width property, 391–392, 405, 560–562  
 Width value, 600  
 wildcards, **736**, 775  
 windows, 109  
 menu strip, 601–604  
 Windows class, 608  
 Windows Form Designer, 106, 626  
 Windows Forms applications and inheritance, 487  
 Windows Forms project, 563  
 adding database, 747  
 WindowsFormsApplication1 application, 104–105, 105  
 Work() method, 480  
 worker1 object, 416  
 worker2 object, 416  
 Write access, 684–685  
 Write() method, **12**, 40, 53, 55–56, 87–88, 211, 261, 683, 685  
 WriteLine() method, **11**–12, 19, 40, 52–53, 55–56, 63, 68, 70, 127, 155, 210–211, 284–285, 288, 296, 448, 539, 683, 685, 688–689, 691  
 writer object, 685  
 WriteSequentialFile class, 689  
 WriteSomeText program, 685–686  
 writing  
 data to sequential access files, 687–690  
 to files, **675**, 684, 720  
 text files, 703–707

**X**

x variable, 328–329, 331–332, 744  
 XML. *See* eXtensible Markup Language (XML)  
 XML-documentation format  
 comments (<>), **18**, 41  
 XML files, 687  
 XmlTextReader class, 687  
 XmlTextWriter class, 687  
 XxxException placeholder, 512–517

**Y**

YB. *See* yottabytes (YB)  
 year field, 460  
 Year property, 460  
 year variable, 168, 170  
 yearsOfService variable, 163  
 yesButton\_Click() method, 230  
 yield contextual keyword, 381  
 yield identifier, 15  
 yottabytes (YB), 795  
 yourAge variable, 51  
 yourAnniversaryParty identifier, 371  
 yourSalary variable, 51

**Z**

ZB. *See* zettabytes (ZB)  
 zero variable, 509  
 zettabytes (ZB), 795