



DEITEL® DEVELOPER SERIES

SIXTH EDITION

# C# 6

## for Programmers

*Use with*  
**Windows® 7, 8 or 10**

PAUL DEITEL • HARVEY DEITEL

FREE SAMPLE CHAPTER

SHARE WITH OTHERS



**C# 6 FOR PROGRAMMERS**  
**SIXTH EDITION**  
DEITEL® DEVELOPER SERIES

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Microsoft and/or its respective suppliers make no representations about the suitability of the information contained in the documents and related graphics published as part of the services for any purpose. All such documents and related graphics are provided “as is” without warranty of any kind. Microsoft and/or its respective suppliers hereby disclaim all warranties and conditions with regard to this information, including all warranties and conditions of merchantability, whether express, implied or statutory, fitness for a particular purpose, title and non-infringement. In no event shall Microsoft and/or its respective suppliers be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of information available from the services.

The documents and related graphics contained herein could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein. Microsoft and/or its respective suppliers may make improvements and/or changes in the product(s) and/or the program(s) described herein at any time. Partial screen shots may be viewed in full within the software version specified.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests.

For government sales inquiries, please contact [governmentsales@pearsoned.com](mailto:governmentsales@pearsoned.com).

For questions about sales outside the U.S., please contact [intlcs@pearson.com](mailto:intlcs@pearson.com).

Visit us on the Web: [informit.com/ph](http://informit.com/ph)

Library of Congress Control Number: 2016946157

Copyright © 2017 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit [www.pearsoned.com/permissions/](http://www.pearsoned.com/permissions/).

ISBN-13: 978-0-13-459632-7

ISBN-10: 0-13-459632-3

Text printed in the United States at RR Donnelley in Crawfordsville, Indiana.

First printing, August 2016

**C# 6 FOR PROGRAMMERS**  
**SIXTH EDITION**  
**DEITEL® DEVELOPER SERIES**

**Paul Deitel**  
*Deitel & Associates, Inc.*

**Harvey Deitel**  
*Deitel & Associates, Inc.*

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town  
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City  
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

# Deitel® Series Page

---

## Deitel® Developer Series

Android™ 6 for Programmers: An App-Driven Approach, 3/E  
C for Programmers with an Introduction to C11  
C++11 for Programmers  
C# 6 for Programmers  
iOS® 8 for Programmers: An App-Driven Approach with Swift™  
Java™ for Programmers, 3/E  
JavaScript for Programmers  
Swift™ for Programmers

## How To Program Series

Android™ How to Program, 3/E  
C++ How to Program, 10/E  
C How to Program, 8/E  
Java™ How to Program, Early Objects Version, 10/E  
Java™ How to Program, Late Objects Version, 10/E  
Internet & World Wide Web How to Program, 5/E  
Visual Basic® 2012 How to Program, 6/E  
Visual C#® How to Program, 6/E

## Simply Series

Simply Visual Basic® 2010: An App-Driven Approach, 4/E  
Simply C++: An App-Driven Tutorial Approach

## VitalSource Web Books

<http://bit.ly/DeitelOnVitalSource>  
Android™ How to Program, 2/E and 3/E  
C++ How to Program, 8/E and 9/E  
Java™ How to Program, 9/E and 10/E  
Simply C++: An App-Driven Tutorial Approach  
Simply Visual Basic® 2010: An App-Driven Approach, 4/E  
Visual Basic® 2012 How to Program, 6/E  
Visual C#® 2012 How to Program, 5/E  
Visual C#® How to Program, 6/E

## LiveLessons Video Learning Products

<http://informit.com/deitel>  
Android™ 6 App Development Fundamentals, 3/E  
C++ Fundamentals  
Java™ Fundamentals, 2/E  
C# 6 Fundamentals  
C# 2012 Fundamentals  
iOS® 8 App Development Fundamentals with Swift™, 3/E  
JavaScript Fundamentals  
Swift™ Fundamentals

## REVEL™ Interactive Multimedia

REVEL™ for Deitel Java™

---

To receive updates on Deitel publications, Resource Centers, training courses, partner offers and more, please join the Deitel communities on

- Facebook®—<http://facebook.com/DeitelFan>
- Twitter®—<http://twitter.com/deitel>
- LinkedIn®—<http://linkedin.com/company/deitel-&-associates>
- YouTube™—<http://youtube.com/DeitelTV>
- Google+™—<http://google.com/+DeitelFan>

and register for the free *Deitel® Buzz Online* e-mail newsletter at:

<http://www.deitel.com/newsletter/subscribe.html>

To communicate with the authors, send e-mail to:

[deitel@deitel.com](mailto:deitel@deitel.com)

For information on programming-languages corporate training seminars offered by Deitel & Associates, Inc. worldwide, write to [deitel@deitel.com](mailto:deitel@deitel.com) or visit:

<http://www.deitel.com/training/>

For continuing updates on Pearson/Deitel publications visit:

<http://www.deitel.com>  
<http://www.pearsonhighered.com/deitel/>

Visit the Deitel Resource Centers, which will help you master programming languages, software development, Android™ and iOS® app development, and Internet- and web-related topics:

<http://www.deitel.com/ResourceCenters.html>

## Trademarks

DEITEL and the double-thumbs-up bug are registered trademarks of Deitel and Associates, Inc.

Microsoft® and Windows® are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.

UNIX is a registered trademark of The Open Group.

Throughout this book, trademarks are used. Rather than put a trademark symbol in every occurrence of a trademarked name, we state that we are using the names in an editorial fashion only and to the benefit of the trademark owner, with no intention of infringement of the trademark.

*In memory of William Siebert, Professor Emeritus of  
Electrical Engineering and Computer Science at MIT:*

*Your use of visualization techniques in  
your Signals and Systems lectures inspired  
the way generations of engineers, computer  
scientists, educators and authors present  
their work.*

*Harvey and Paul Deitel*

# Contents

**Preface** **xxi**

**Before You Begin** **xxxii**

**1 Introduction** **1**

1.1	Introduction	2
1.2	Object Technology: A Brief Review	2
1.3	C#	5
1.3.1	Object-Oriented Programming	5
1.3.2	Event-Driven Programming	6
1.3.3	Visual Programming	6
1.3.4	Generic and Functional Programming	6
1.3.5	An International Standard	6
1.3.6	C# on Non-Windows Platforms	6
1.3.7	Internet and Web Programming	7
1.3.8	Asynchronous Programming with <code>async</code> and <code>await</code>	7
1.4	Microsoft's .NET	7
1.4.1	.NET Framework	7
1.4.2	Common Language Runtime	7
1.4.3	Platform Independence	8
1.4.4	Language Interoperability	8
1.5	Microsoft's Windows® Operating System	8
1.6	Visual Studio Integrated Development Environment	10
1.7	Painter Test-Drive in Visual Studio Community	10

**2 Introduction to Visual Studio and Visual Programming** **15**

2.1	Introduction	16
2.2	Overview of the Visual Studio Community 2015 IDE	16
2.2.1	Introduction to Visual Studio Community 2015	16
2.2.2	Visual Studio Themes	17
2.2.3	Links on the Start Page	17
2.2.4	Creating a New Project	18
2.2.5	<b>New Project</b> Dialog and Project Templates	19
2.2.6	Forms and Controls	20



2.3	Menu Bar and Toolbar	21
2.4	Navigating the Visual Studio IDE	24
2.4.1	<b>Solution Explorer</b>	25
2.4.2	<b>Toolbox</b>	26
2.4.3	<b>Properties Window</b>	26
2.5	Help Menu and Context-Sensitive Help	28
2.6	Visual Programming: Creating a Simple App that Displays Text and an Image	29
2.7	Wrap-Up	38
2.8	Web Resources	39

## 3 Introduction to C# App Programming 40

3.1	Introduction	41
3.2	Simple App: Displaying a Line of Text	41
3.2.1	Comments	42
3.2.2	using Directive	43
3.2.3	Blank Lines and Whitespace	43
3.2.4	Class Declaration	43
3.2.5	Main Method	46
3.2.6	Displaying a Line of Text	46
3.2.7	Matching Left ( { ) and Right ( } ) Braces	47
3.3	Creating a Simple App in Visual Studio	47
3.3.1	Creating the Console App	47
3.3.2	Changing the Name of the App File	48
3.3.3	Writing Code and Using <i>IntelliSense</i>	49
3.3.4	Compiling and Running the App	51
3.3.5	Errors, Error Messages and the <b>Error List</b> Window	51
3.4	Modifying Your Simple C# App	52
3.4.1	Displaying a Single Line of Text with Multiple Statements	52
3.4.2	Displaying Multiple Lines of Text with a Single Statement	53
3.5	String Interpolation	55
3.6	Another C# App: Adding Integers	56
3.6.1	Declaring the int Variable number1	57
3.6.2	Declaring Variables number2 and sum	57
3.6.3	Prompting the User for Input	58
3.6.4	Reading a Value into Variable number1	58
3.6.5	Prompting the User for Input and Reading a Value into number2	59
3.6.6	Summing number1 and number2	59
3.6.7	Displaying the sum with string Interpolation	59
3.6.8	Performing Calculations in Output Statements	59
3.7	Arithmetic	59
3.7.1	Arithmetic Expressions in Straight-Line Form	60
3.7.2	Parentheses for Grouping Subexpressions	60
3.7.3	Rules of Operator Precedence	60
3.8	Decision Making: Equality and Relational Operators	61
3.9	Wrap-Up	65

<b>4</b>	<b>Introduction to Classes, Objects, Methods and strings</b>	<b>67</b>
4.1	Introduction	68
4.2	Test-Driving an Account Class	69
4.2.1	Instantiating an Object—Keyword <code>new</code>	69
4.2.2	Calling Class <code>Account</code> 's <code>GetName</code> Method	70
4.2.3	Inputting a Name from the User	70
4.2.4	Calling Class <code>Account</code> 's <code>SetName</code> Method	71
4.3	<code>Account</code> Class with an Instance Variable and <i>Set</i> and <i>Get</i> Methods	71
4.3.1	<code>Account</code> Class Declaration	71
4.3.2	Keyword <code>class</code> and the Class Body	72
4.3.3	Instance Variable name of Type <code>string</code>	72
4.3.4	<code>SetName</code> Method	73
4.3.5	<code>GetName</code> Method	75
4.3.6	Access Modifiers <code>private</code> and <code>public</code>	75
4.3.7	<code>Account</code> UML Class Diagram	76
4.4	Creating, Compiling and Running a Visual C# Project with Two Classes	77
4.5	Software Engineering with <i>Set</i> and <i>Get</i> Methods	78
4.6	<code>Account</code> Class with a Property Rather Than <i>Set</i> and <i>Get</i> Methods	79
4.6.1	Class <code>AccountTest</code> Using <code>Account</code> 's Name Property	79
4.6.2	<code>Account</code> Class with an Instance Variable and a Property	81
4.6.3	<code>Account</code> UML Class Diagram with a Property	83
4.7	Auto-Implemented Properties	83
4.8	<code>Account</code> Class: Initializing Objects with Constructors	84
4.8.1	Declaring an <code>Account</code> Constructor for Custom Object Initialization	84
4.8.2	Class <code>AccountTest</code> : Initializing <code>Account</code> Objects When They're Created	85
4.9	<code>Account</code> Class with a Balance; Processing Monetary Amounts	87
4.9.1	<code>Account</code> Class with a <code>decimal</code> balance Instance Variable	87
4.9.2	<code>AccountTest</code> Class That Uses <code>Account</code> Objects with Balances	90
4.10	Wrap-Up	93
<b>5</b>	<b>Control Statements: Part I</b>	<b>95</b>
5.1	Introduction	96
5.2	Control Structures	96
5.2.1	Sequence Structure	97
5.2.2	Selection Statements	98
5.2.3	Iteration Statements	98
5.2.4	Summary of Control Statements	99
5.3	<code>if</code> Single-Selection Statement	99
5.4	<code>if...else</code> Double-Selection Statement	100
5.4.1	Nested <code>if...else</code> Statements	101
5.4.2	Dangling- <code>else</code> Problem	102

5.4.3	Blocks	102
5.4.4	Conditional Operator (?:)	103
5.5	Student Class: Nested <code>if...else</code> Statements	103
5.6	<code>while</code> Iteration Statement	106
5.7	Counter-Controlled Iteration	107
5.7.1	Implementing Counter-Controlled Iteration	108
5.7.2	Integer Division and Truncation	110
5.8	Sentinel-Controlled Iteration	110
5.8.1	Implementing Sentinel-Controlled Iteration	110
5.8.2	Program Logic for Sentinel-Controlled Iteration	112
5.8.3	Braces in a <code>while</code> Statement	113
5.8.4	Converting Between Simple Types Explicitly and Implicitly	113
5.8.5	Formatting Floating-Point Numbers	114
5.9	Nested Control Statements	114
5.10	Compound Assignment Operators	117
5.11	Increment and Decrement Operators	118
5.11.1	Prefix Increment vs. Postfix Increment	119
5.11.2	Simplifying Increment Statements	120
5.11.3	Operator Precedence and Associativity	120
5.12	Simple Types	121
5.13	Wrap-Up	121

## 6 Control Statements: Part 2 123

6.1	Introduction	124
6.2	Essentials of Counter-Controlled Iteration	124
6.3	<code>for</code> Iteration Statement	125
6.3.1	A Closer Look at the <code>for</code> Statement's Header	126
6.3.2	General Format of a <code>for</code> Statement	126
6.3.3	Scope of a <code>for</code> Statement's Control Variable	127
6.3.4	Expressions in a <code>for</code> Statement's Header Are Optional	127
6.3.5	UML Activity Diagram for the <code>for</code> Statement	127
6.4	App: Summing Even Integers	128
6.5	App: Compound-Interest Calculations	129
6.5.1	Performing the Interest Calculations with Math Method <code>pow</code>	130
6.5.2	Formatting with Field Widths and Alignment	131
6.5.3	Caution: Do Not Use <code>float</code> or <code>double</code> for Monetary Amounts	131
6.6	<code>do...while</code> Iteration Statement	132
6.7	<code>switch</code> Multiple-Selection Statement	133
6.7.1	Using a <code>switch</code> Statement to Count A, B, C, D and F Grades	133
6.7.2	<code>switch</code> Statement UML Activity Diagram	138
6.7.3	Notes on the Expression in Each case of a <code>switch</code>	138
6.8	Class <code>AutoPolicy</code> Case Study: strings in <code>switch</code> Statements	139
6.9	<code>break</code> and <code>continue</code> Statements	141
6.9.1	<code>break</code> Statement	141
6.9.2	<code>continue</code> Statement	142
6.10	Logical Operators	143

6.10.1	Conditional AND (&&) Operator	143
6.10.2	Conditional OR (  ) Operator	144
6.10.3	Short-Circuit Evaluation of Complex Conditions	145
6.10.4	Boolean Logical AND (&) and Boolean Logical OR ( ) Operators	145
6.10.5	Boolean Logical Exclusive OR (^)	145
6.10.6	Logical Negation (!) Operator	146
6.10.7	Logical Operators Example	146
6.11	Wrap-Up	149

## **7** **Methods: A Deeper Look** **150**

7.1	Introduction	151
7.2	Packaging Code in C#	152
7.3	static Methods, static Variables and Class Math	152
7.3.1	Math Class Methods	153
7.3.2	Math Class Constants PI and E	154
7.3.3	Why Is Main Declared static?	154
7.3.4	Additional Comments About Main	155
7.4	Methods with Multiple Parameters	155
7.4.1	Keyword static	157
7.4.2	Method Maximum	157
7.4.3	Assembling strings with Concatenation	157
7.4.4	Breaking Apart Large string Literals	158
7.4.5	When to Declare Variables as Fields	159
7.4.6	Implementing Method Maximum by Reusing Method Math.Max	159
7.5	Notes on Using Methods	159
7.6	Argument Promotion and Casting	160
7.6.1	Promotion Rules	161
7.6.2	Sometimes Explicit Casts Are Required	161
7.7	The .NET Framework Class Library	162
7.8	Case Study: Random-Number Generation	164
7.8.1	Creating an Object of Type Random	164
7.8.2	Generating a Random Integer	164
7.8.3	Scaling the Random-Number Range	165
7.8.4	Shifting Random-Number Range	165
7.8.5	Combining Shifting and Scaling	165
7.8.6	Rolling a Six-Sided Die	165
7.8.7	Scaling and Shifting Random Numbers	168
7.8.8	Repeatability for Testing and Debugging	168
7.9	Case Study: A Game of Chance; Introducing Enumerations	169
7.9.1	Method RollDice	172
7.9.2	Method Main's Local Variables	172
7.9.3	enum Type Status	172
7.9.4	The First Roll	173
7.9.5	enum Type DiceNames	173
7.9.6	Underlying Type of an enum	173
7.9.7	Comparing Integers and enum Constants	173

7.10	Scope of Declarations	174
7.11	Method-Call Stack and Activation Records	177
7.11.1	Method-Call Stack	177
7.11.2	Stack Frames	177
7.11.3	Local Variables and Stack Frames	178
7.11.4	Stack Overflow	178
7.11.5	Method-Call Stack in Action	178
7.12	Method Overloading	181
7.12.1	Declaring Overloaded Methods	181
7.12.2	Distinguishing Between Overloaded Methods	182
7.12.3	Return Types of Overloaded Methods	182
7.13	Optional Parameters	183
7.14	Named Parameters	184
7.15	C# 6 Expression-Bodied Methods and Properties	185
7.16	Recursion	186
7.16.1	Base Cases and Recursive Calls	186
7.16.2	Recursive Factorial Calculations	186
7.16.3	Implementing Factorial Recursively	187
7.17	Value Types vs. Reference Types	189
7.18	Passing Arguments By Value and By Reference	190
7.18.1	ref and out Parameters	191
7.18.2	Demonstrating ref, out and Value Parameters	192
7.19	Wrap-Up	194
<b>8</b>	<b>Arrays; Introduction to Exception Handling</b>	<b>195</b>
8.1	Introduction	196
8.2	Arrays	197
8.3	Declaring and Creating Arrays	198
8.4	Examples Using Arrays	199
8.4.1	Creating and Initializing an Array	199
8.4.2	Using an Array Initializer	200
8.4.3	Calculating a Value to Store in Each Array Element	201
8.4.4	Summing the Elements of an Array	202
8.4.5	Iterating Through Arrays with foreach	203
8.4.6	Using Bar Charts to Display Array Data Graphically; Introducing Type Inference with var	205
8.4.7	Using the Elements of an Array as Counters	207
8.5	Using Arrays to Analyze Survey Results; Intro to Exception Handling	208
8.5.1	Summarizing the Results	210
8.5.2	Exception Handling: Processing the Incorrect Response	211
8.5.3	The try Statement	211
8.5.4	Executing the catch Block	211
8.5.5	Message Property of the Exception Parameter	211
8.6	Case Study: Card Shuffling and Dealing Simulation	212
8.6.1	Class Card and Getter-Only Auto-Implemented Properties	212
8.6.2	Class DeckOfCards	213

8.6.3	Shuffling and Dealing Cards	215
8.7	Passing Arrays and Array Elements to Methods	216
8.8	Case Study: GradeBook Using an Array to Store Grades	219
8.9	Multidimensional Arrays	225
8.9.1	Rectangular Arrays	225
8.9.2	Jagged Arrays	226
8.9.3	Two-Dimensional Array Example: Displaying Element Values	227
8.10	Case Study: GradeBook Using a Rectangular Array	230
8.11	Variable-Length Argument Lists	236
8.12	Using Command-Line Arguments	237
8.13	(Optional) Passing Arrays by Value and by Reference	240
8.14	Wrap-Up	244

## **9 Introduction to LINQ and the List Collection 245**

9.1	Introduction	246
9.2	Querying an Array of <code>int</code> Values Using LINQ	247
9.2.1	The <code>from</code> Clause	249
9.2.2	The <code>where</code> Clause	250
9.2.3	The <code>select</code> Clause	250
9.2.4	Iterating Through the Results of the LINQ Query	250
9.2.5	The <code>orderby</code> Clause	250
9.2.6	Interface <code>IEnumerable&lt;T&gt;</code>	251
9.3	Querying an Array of <code>Employee</code> Objects Using LINQ	251
9.3.1	Accessing the Properties of a LINQ Query's Range Variable	255
9.3.2	Sorting a LINQ Query's Results by Multiple Properties	255
9.3.3	<code>Any</code> , <code>First</code> and <code>Count</code> Extension Methods	255
9.3.4	Selecting a Property of an Object	255
9.3.5	Creating New Types in the <code>select</code> Clause of a LINQ Query	255
9.4	Introduction to Collections	256
9.4.1	<code>List&lt;T&gt;</code> Collection	256
9.4.2	Dynamically Resizing a <code>List&lt;T&gt;</code> Collection	257
9.5	Querying the Generic <code>List</code> Collection Using LINQ	261
9.5.1	The <code>let</code> Clause	263
9.5.2	Deferred Execution	263
9.5.3	Extension Methods <code>ToArray</code> and <code>ToList</code>	263
9.5.4	Collection Initializers	263
9.6	Wrap-Up	264
9.7	Deitel LINQ Resource Center	264

## **10 Classes and Objects: A Deeper Look 265**

10.1	Introduction	266
10.2	Time Class Case Study; Throwing Exceptions	266
10.2.1	<code>Time1</code> Class Declaration	267
10.2.2	Using Class <code>Time1</code>	268
10.3	Controlling Access to Members	270

10.4	Referring to the Current Object’s Members with the <code>this</code> Reference	271
10.5	Time Class Case Study: Overloaded Constructors	273
10.5.1	Class <code>Time2</code> with Overloaded Constructors	273
10.5.2	Using Class <code>Time2</code> ’s Overloaded Constructors	277
10.6	Default and Parameterless Constructors	279
10.7	Composition	280
10.7.1	Class <code>Date</code>	280
10.7.2	Class <code>Employee</code>	282
10.7.3	Class <code>EmployeeTest</code>	283
10.8	Garbage Collection and Destructors	284
10.9	<code>static</code> Class Members	284
10.10	<code>readonly</code> Instance Variables	288
10.11	<b>Class View</b> and <b>Object Browser</b>	289
10.11.1	Using the <b>Class View</b> Window	289
10.11.2	Using the <b>Object Browser</b>	290
10.12	Object Initializers	291
10.13	Operator Overloading; Introducing <code>struct</code>	291
10.13.1	Creating Value Types with <code>struct</code>	292
10.13.2	Value Type <code>ComplexNumber</code>	292
10.13.3	Class <code>ComplexTest</code>	294
10.14	Time Class Case Study: Extension Methods	295
10.15	Wrap-Up	298

## **11 Object-Oriented Programming: Inheritance**    **299**

11.1	Introduction	300
11.2	Base Classes and Derived Classes	301
11.3	<code>protected</code> Members	303
11.4	Relationship between Base Classes and Derived Classes	304
11.4.1	Creating and Using a <code>CommissionEmployee</code> Class	305
11.4.2	Creating a <code>BasePlusCommissionEmployee</code> Class without Using Inheritance	309
11.4.3	Creating a <code>CommissionEmployee–BasePlusCommissionEmployee</code> Inheritance Hierarchy	314
11.4.4	<code>CommissionEmployee–BasePlusCommissionEmployee</code> Inheritance Hierarchy Using <code>protected</code> Instance Variables	317
11.4.5	<code>CommissionEmployee–BasePlusCommissionEmployee</code> Inheritance Hierarchy Using <code>private</code> Instance Variables	320
11.5	Constructors in Derived Classes	324
11.6	Software Engineering with Inheritance	324
11.7	Class object	325
11.8	Wrap-Up	326

## **12 OOP: Polymorphism and Interfaces**    **327**

12.1	Introduction	328
12.2	Polymorphism Examples	330

12.3	Demonstrating Polymorphic Behavior	331
12.4	Abstract Classes and Methods	334
12.5	Case Study: Payroll System Using Polymorphism	336
12.5.1	Creating Abstract Base Class <code>Employee</code>	337
12.5.2	Creating Concrete Derived Class <code>SalariesEmployee</code>	339
12.5.3	Creating Concrete Derived Class <code>HourlyEmployee</code>	341
12.5.4	Creating Concrete Derived Class <code>CommissionEmployee</code>	342
12.5.5	Creating Indirect Concrete Derived Class <code>BasePlusCommissionEmployee</code>	344
12.5.6	Polymorphic Processing, Operator <code>is</code> and Downcasting	345
12.5.7	Summary of the Allowed Assignments Between Base-Class and Derived-Class Variables	350
12.6	<code>sealed</code> Methods and Classes	351
12.7	Case Study: Creating and Using Interfaces	352
12.7.1	Developing an <code>IPayable</code> Hierarchy	353
12.7.2	Declaring Interface <code>IPayable</code>	355
12.7.3	Creating Class <code>Invoice</code>	355
12.7.4	Modifying Class <code>Employee</code> to Implement Interface <code>IPayable</code>	357
12.7.5	Using Interface <code>IPayable</code> to Process Invoices and Employees Polymorphically	358
12.7.6	Common Interfaces of the .NET Framework Class Library	360
12.8	Wrap-Up	361
<b>13</b>	<b>Exception Handling: A Deeper Look</b>	<b>362</b>
13.1	Introduction	363
13.2	Example: Divide by Zero without Exception Handling	364
13.2.1	Dividing By Zero	365
13.2.2	Enter a Non-Numeric Denominator	366
13.2.3	Unhandled Exceptions Terminate the App	366
13.3	Example: Handling <code>DivideByZeroExceptions</code> and <code>FormatExceptions</code>	367
13.3.1	Enclosing Code in a <code>try</code> Block	369
13.3.2	Catching Exceptions	369
13.3.3	Uncaught Exceptions	370
13.3.4	Termination Model of Exception Handling	371
13.3.5	Flow of Control When Exceptions Occur	371
13.4	.NET Exception Hierarchy	372
13.4.1	Class <code>SystemException</code>	372
13.4.2	Which Exceptions Might a Method Throw?	373
13.5	<code>finally</code> Block	374
13.5.1	Moving Resource-Release Code to a <code>finally</code> Block	374
13.5.2	Demonstrating the <code>finally</code> Block	375
13.5.3	Throwing Exceptions Using the <code>throw</code> Statement	379
13.5.4	Rethrowing Exceptions	379
13.5.5	Returning After a <code>finally</code> Block	380
13.6	The <code>using</code> Statement	381



13.7	Exception Properties	382
13.7.1	Property InnerException	382
13.7.2	Other Exception Properties	383
13.7.3	Demonstrating Exception Properties and Stack Unwinding	383
13.7.4	Throwing an Exception with an InnerException	385
13.7.5	Displaying Information About the Exception	386
13.8	User-Defined Exception Classes	386
13.9	Checking for null References; Introducing C# 6's ?. Operator	390
13.9.1	Null-Conditional Operator (?.)	390
13.9.2	Revisiting Operators is and as	391
13.9.3	Nullable Types	391
13.9.4	Null Coalescing Operator (??)	392
13.10	Exception Filters and the C# 6 when Clause	392
13.11	Wrap-Up	393

## **14 Graphical User Interfaces with Windows Forms: Part 1** **394**

14.1	Introduction	395
14.2	Windows Forms	396
14.3	Event Handling	398
14.3.1	A Simple Event-Driven GUI	399
14.3.2	Auto-Generated GUI Code	400
14.3.3	Delegates and the Event-Handling Mechanism	403
14.3.4	Another Way to Create Event Handlers	404
14.3.5	Locating Event Information	405
14.4	Control Properties and Layout	406
14.4.1	Anchoring and Docking	407
14.4.2	Using Visual Studio To Edit a GUI's Layout	409
14.5	Labels, TextBoxes and Buttons	410
14.6	GroupBoxes and Panels	413
14.7	CheckBoxes and RadioButtons	416
14.7.1	CheckBoxes	416
14.7.2	Combining Font Styles with Bitwise Operators	418
14.7.3	RadioButtons	419
14.8	PictureBoxes	424
14.9	ToolTips	426
14.10	NumericUpDown Control	428
14.11	Mouse-Event Handling	430
14.12	Keyboard-Event Handling	433
14.13	Wrap-Up	436

## **15 Graphical User Interfaces with Windows Forms: Part 2** **438**

15.1	Introduction	439
------	--------------	-----

15.2	Menus	439
15.3	MonthCalendar Control	449
15.4	DateTimePicker Control	450
15.5	LinkLabel Control	453
15.6	ListBox Control	456
15.7	CheckedListBox Control	461
15.8	ComboBox Control	464
15.9	TreeView Control	468
15.10	ListView Control	474
15.11	TabControl Control	480
15.12	Multiple Document Interface (MDI) Windows	484
15.13	Visual Inheritance	492
15.14	User-Defined Controls	497
15.15	Wrap-Up	500

## **16 Strings and Characters: A Deeper Look** **502**

16.1	Introduction	503
16.2	Fundamentals of Characters and Strings	504
16.3	string Constructors	505
16.4	string Indexer, Length Property and CopyTo Method	506
16.5	Comparing strings	507
16.6	Locating Characters and Substrings in strings	511
16.7	Extracting Substrings from strings	514
16.8	Concatenating strings	515
16.9	Miscellaneous string Methods	515
16.10	Class StringBuilder	517
	16.11 Length and Capacity Properties, EnsureCapacity Method and Indexer of Class StringBuilder	518
	16.12 Append and AppendFormat Methods of Class StringBuilder	520
	16.13 Insert, Remove and Replace Methods of Class StringBuilder	522
16.14	Char Methods	525
16.15	Introduction to Regular Expressions (Online)	527
16.16	Wrap-Up	527

## **17 Files and Streams** **529**

17.1	Introduction	530
17.2	Files and Streams	530
17.3	Creating a Sequential-Access Text File	531
17.4	Reading Data from a Sequential-Access Text File	540
17.5	Case Study: Credit-Inquiry Program	544
17.6	Serialization	549
17.7	Creating a Sequential-Access File Using Object Serialization	550
17.8	Reading and Deserializing Data from a Binary File	554
17.9	Classes File and Directory	557
	17.9.1 Demonstrating Classes File and Directory	558

17.9.2	Searching Directories with LINQ	561
17.10	Wrap-Up	565

## 18 Generics 567

18.1	Introduction	568
18.2	Motivation for Generic Methods	569
18.3	Generic-Method Implementation	571
18.4	Type Constraints	574
18.4.1	IComparable<T> Interface	574
18.4.2	Specifying Type Constraints	574
18.5	Overloading Generic Methods	577
18.6	Generic Classes	577
18.7	Wrap-Up	587

## 19 Generic Collections; Functional Programming with LINQ/PLINQ 588

19.1	Introduction	589
19.2	Collections Overview	590
19.3	Class Array and Enumerators	593
19.3.1	C# 6 using static Directive	595
19.3.2	Class UsingArray's static Fields	596
19.3.3	Array Method Sort	596
19.3.4	Array Method Copy	596
19.3.5	Array Method BinarySearch	596
19.3.6	Array Method GetEnumerator and Interface IEnumerator	596
19.3.7	Iterating Over a Collection with foreach	597
19.3.8	Array Methods Clear, IndexOf, LastIndexOf and Reverse	597
19.4	Dictionary Collections	597
19.4.1	Dictionary Fundamentals	598
19.4.2	Using the SortedDictionary Collection	599
19.5	Generic LinkedList Collection	603
19.6	C# 6 Null Conditional Operator ?[]	607
19.7	C# 6 Dictionary Initializers and Collection Initializers	608
19.8	Delegates	608
19.8.1	Declaring a Delegate Type	610
19.8.2	Declaring a Delegate Variable	610
19.8.3	Delegate Parameters	611
19.8.4	Passing a Method Name Directly to a Delegate Parameter	611
19.9	Lambda Expressions	611
19.9.1	Expression Lambdas	613
19.9.2	Assigning Lambdas to Delegate Variables	614
19.9.3	Explicitly Typed Lambda Parameters	614
19.9.4	Statement Lambdas	614
19.10	Introduction to Functional Programming	614
19.11	Functional Programming with LINQ Method-Call Syntax and Lambdas	616

19.11.1	LINQ Extension Methods Min, Max, Sum and Average	619
19.11.2	Aggregate Extension Method for Reduction Operations	619
19.11.3	The Where Extension Method for Filtering Operations	621
19.11.4	Select Extension Method for Mapping Operations	622
19.12	PLINQ: Improving LINQ to Objects Performance with Multicore	622
19.13	(Optional) Covariance and Contravariance for Generic Types	626
19.14	Wrap-Up	628

## **20 Databases and LINQ** **629**

20.1	Introduction	630
20.2	Relational Databases	631
20.3	A Books Database	632
20.4	LINQ to Entities and the ADO.NET Entity Framework	636
20.5	Querying a Database with LINQ	637
20.5.1	Creating the ADO.NET Entity Data Model Class Library	639
20.5.2	Creating a Windows Forms Project and Configuring It to Use the Entity Data Model	643
20.5.3	Data Bindings Between Controls and the Entity Data Model	645
20.6	Dynamically Binding Query Results	651
20.6.1	Creating the <b>Display Query Results</b> GUI	652
20.6.2	Coding the <b>Display Query Results</b> App	653
20.7	Retrieving Data from Multiple Tables with LINQ	655
20.8	Creating a Master/Detail View App	661
20.8.1	Creating the Master/Detail GUI	661
20.8.2	Coding the Master/Detail App	663
20.9	Address Book Case Study	664
20.9.1	Creating the <b>Address Book</b> App's GUI	666
20.9.2	Coding the <b>Address Book</b> App	667
20.10	Tools and Web Resources	671
20.11	Wrap-Up	671

## **21 Asynchronous Programming with async and await** **672**

21.1	Introduction	673
21.2	Basics of async and await	675
21.2.1	async Modifier	675
21.2.2	await Expression	675
21.2.3	async, await and Threads	675
21.3	Executing an Asynchronous Task from a GUI App	676
21.3.1	Performing a Task Asynchronously	676
21.3.2	Method calculateButton_Click	678
21.3.3	Task Method Run: Executing Asynchronously in a Separate Thread	679
21.3.4	awaiting the Result	679
21.3.5	Calculating the Next Fibonacci Value Synchronously	679
21.4	Sequential Execution of Two Compute-Intensive Tasks	680

21.5	Asynchronous Execution of Two Compute-Intensive Tasks	682
21.5.1	awaiting Multiple Tasks with Task Method WhenAll	685
21.5.2	Method StartFibonacci	686
21.5.3	Modifying a GUI from a Separate Thread	686
21.5.4	awaiting One of Several Tasks with Task Method WhenAny	686
21.6	Invoking a Flickr Web Service Asynchronously with HttpClient	687
21.6.1	Using Class HttpClient to Invoke a Web Service	691
21.6.2	Invoking the Flickr Web Service's flickr.photos.search Method	691
21.6.3	Processing the XML Response	692
21.6.4	Binding the Photo Titles to the ListBox	693
21.6.5	Asynchronously Downloading an Image's Bytes	694
21.7	Displaying an Asynchronous Task's Progress	694
21.8	Wrap-Up	698

**A Operator Precedence Chart 700**

**B Simple Types 702**

**C ASCII Character Set 704**

**Index 705**

# Preface

Welcome to the world of leading-edge software development with Microsoft's® Visual C#® programming language. *C# 6 for Programmers, 6/e* is based on C# 6 and related Microsoft software technologies.<sup>1</sup> You'll be using the .NET platform and the Visual Studio® Integrated Development Environment on which you'll conveniently write, test and debug your applications and run them on Windows® devices. The Windows operating system runs on desktop and notebook computers, mobile phones and tablets, game systems and a great variety of devices associated with the emerging "Internet of Things." We believe that this book will give you an informative, engaging, challenging and entertaining introduction to C#.

You'll study C# in the context of four of today's most popular programming paradigms:

- object-oriented programming,
- structured programming,
- generic programming and
- functional programming (new in this edition).

If you haven't already done so, please read the back cover and check out the additional reviewer comments on the inside back cover—these capture the essence of the book concisely. In this Preface we provide more detail.

The book is loaded with "live-code" examples—most new concepts are presented in the context of complete working C# apps, followed by one or more executions showing program inputs and outputs. In the few cases where we show a code snippet, to ensure correctness first we tested it in a working program then copied the code and pasted it into the book. We include a broad range of example apps selected from business, education, computer science, personal utilities, mathematics, simulation, game playing, graphics and many other areas. We also provide abundant tables, line drawings and UML diagrams.

Read the Before You Begin section after this Preface for instructions on setting up your computer to run the 170+ code examples and to enable you to develop your own C# apps. The source code for all of the book's examples is available at

<http://www.deitel.com/books/CSharp6FP>

Use the source code we provide to compile and run each program as you study it—this will help you master C# and related Microsoft technologies faster and at a deeper level.

1. At the time of this writing, Microsoft has not yet released the official C# 6 Specification. To view an unofficial copy, visit <https://github.com/1jw1004/csharp6spec/blob/gh-pages/README.md>

## Contacting the Authors

As you read the book, if you have a question, we're easy to reach at

`deitel@deitel.com`

We'll respond promptly.

## Join the Deitel & Associates, Inc. Social Media Communities

For book updates, visit

`http://www.deitel.com/books/CSharp6FP`

subscribe to the *Deitel*<sup>®</sup> *Buzz Online* newsletter

`http://www.deitel.com/newsletter/subscribe.html`

and join the conversation on

- Facebook<sup>®</sup>—`http://facebook.com/DeitelFan`
- LinkedIn<sup>®</sup>—`http://linkedin.com/company/deitel-&-associates`
- YouTube<sup>®</sup>—`http://youtube.com/DeitelTV`
- Twitter<sup>®</sup>—`http://twitter.com/Deitel`
- Instagram<sup>®</sup>—`http://instagram.com/DeitelFan`
- Google+<sup>™</sup>—`http://google.com/+DeitelFan`

## New C# 6 Features

6 We introduce key new C# 6 language features throughout the book (Fig. 1)—each defining occurrence is marked with a “6” margin icon as shown next to this paragraph.

C# 6 new language feature	First introduced in
string interpolation	Section 3.5
expression-bodied methods and get accessors	Section 7.15
auto-implemented property initializers	Section 8.6.1
getter-only auto-implemented properties	Section 8.6.1
nameof operator	Section 10.5.1
null-conditional operator (?.)	Section 13.9.1
when clause for exception filtering	Section 13.10
using static directive	Section 19.3.1
null conditional operator (?[])	Section 19.6
collection initializers for any collection with an Add extension method	Section 19.7
index initializers	Section 19.7

**Fig. 1** | C# 6 new language features.

## A Tour of the Book

Here's a quick walkthrough of the book's key features.

### *Introduction to Visual C# and Visual Studio 2015 Community Edition*

The discussions in

- Chapter 1, Introduction
- Chapter 2, Introduction to Visual Studio and Visual Programming

introduce the C# programming language, Microsoft's .NET platform and Visual Programming. The vast majority of the book's examples will run on Windows 7, 8 and 10 using the *Visual Studio 2015 Community* edition with which we test-drive a **Painter** app in Section 1.7. Chapter 1 briefly reviews object-oriented programming terminology and concepts on which the rest of the book depends.

### *Introduction to C# Fundamentals*

The discussions in

- Chapter 3, Introduction to C# App Programming
- Chapter 4, Introduction to Classes, Objects, Methods and strings
- Chapter 5, Control Statements: Part 1
- Chapter 6, Control Statements: Part 2
- Chapter 7, Methods: A Deeper Look
- Chapter 8, Arrays; Introduction to Exception Handling

present rich coverage of C# programming fundamentals (data types, classes, objects, operators, control statements, methods and arrays) through a series of object-oriented programming case studies. Chapter 8 briefly introduces exception handling with an example that demonstrates attempting to access an element outside an array's bounds.

### *Object-Oriented Programming: A Deeper Look*

The discussions in

- Chapter 9, Introduction to LINQ and the List Collection
- Chapter 10, Classes and Objects: A Deeper Look
- Chapter 11, Object-Oriented Programming: Inheritance
- Chapter 12, OOP: Polymorphism and Interfaces
- Chapter 13, Exception Handling: A Deeper Look

provide a deeper look at object-oriented programming, including classes, objects, inheritance, polymorphism, interfaces and exception handling. An online two-chapter case study on designing and implementing the object-oriented software for a simple ATM is described later in this preface.

Chapter 9 introduces Microsoft's Language Integrated Query (LINQ) technology, which provides a uniform syntax for manipulating data from various data sources, such as arrays, collections and, as you'll see in later chapters, databases and XML. Chapter 9 is intentionally simple and brief to encourage readers to begin using LINQ technology early.



Section 9.4 introduces the `List` collection. Later in the book, we take a deeper look at LINQ, using LINQ to Entities (for querying databases) and LINQ to XML.

### *Windows Forms Graphical User Interfaces (GUIs)*

The discussions in

- Chapter 14, Graphical User Interfaces with Windows Forms: Part 1
- Chapter 15, Graphical User Interfaces with Windows Forms: Part 2

present a detailed introduction to building GUIs using Windows Forms. We also use Windows Forms GUIs in several later chapters.

### *Strings and Files*

The discussions in

- Chapter 16, Strings and Characters: A Deeper Look
- Chapter 17, Files and Streams

investigate strings in more detail, and introduce text-file processing and object-serialization for inputting and outputting entire objects.

### *Generics and Generic Collections*

The discussions in

- Chapter 18, Generics
- Chapter 19, Generic Collections; Functional Programming with LINQ/PLINQ

introduce generics and generic collections. Chapter 18 introduces C# generics and demonstrates how to create type-safe generic methods and a type-safe generic class. Rather than “reinventing the wheel,” most C# programmers should use .NET’s built-in searching, sorting and generic collections (prepackaged data structures) capabilities, which are discussed in Chapter 19.

### *Functional Programming with LINQ, PLINQ, Lambdas, Delegates and Immutability*

In addition to generic collections, Chapter 19 now introduces functional programming, showing how to use it with LINQ to Objects to write code more concisely and with fewer bugs than programs written using previous techniques. In Section 19.12, with one additional method call, we demonstrate with timing examples how PLINQ (Parallel LINQ) can improve LINQ to Objects performance substantially on multicore systems.

### *Database with LINQ to Entities and SQL Server*

The discussions in

- Chapter 20, Databases and LINQ

introduce database programming with the ADO.NET Entity Framework, LINQ to Entities and Microsoft’s free version of SQL Server that’s installed with the Visual Studio 2015 Community edition.

### *Asynchronous Programming*

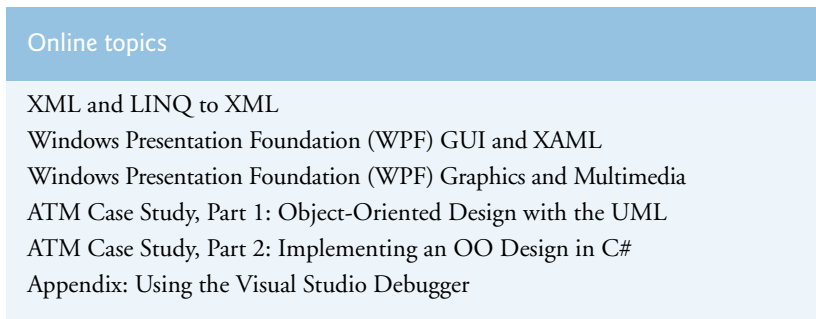
The discussions in

- Chapter 21, Asynchronous Programming with `async` and `await`

show how to take advantage of multicore architectures by writing applications that can process tasks asynchronously, which can improve app performance and GUI responsiveness in apps with long-running or compute-intensive tasks. The `async` modifier and `await` operator greatly simplify asynchronous programming, reduce errors and enable your apps to take advantage of the processing power in today's multicore computers, smartphones and tablets. In this edition, we added a case study that uses the Task Parallel Library (TPL), `async` and `await` in a GUI app—we keep a progress bar moving along in the GUI thread in parallel with a lengthy, compute-intensive calculation in another thread.

## Online Bonus Content

Figure 2 shows online bonus content available with the publication of the book.



**Fig. 2** | Online topics on the *C# 6 for Programmers, 6/e* Companion Website.

### *Accessing the Bonus Content*

To access these materials—and for downloads, updates and corrections as they become available—register your copy of *C# 6 for Programmers, 6/e* at [informit.com](http://informit.com). To register:

1. Go to

<http://informit.com/register>

2. Log in or create an account.
3. Enter the product ISBN—9780134596327—and click **Submit**.

Once you've registered your book, you'll find any available bonus content under **Registered Products**. Here's a quick walkthrough of the initial online content.

### *XML and LINQ to XML*

The Extensible Markup Language (XML), introduced briefly in Chapter 21, is pervasive in the software-development industry, e-business and throughout the .NET platform. XML is required to understand XAML—a Microsoft XML vocabulary that's used to describe graphical user interfaces, graphics and multimedia for Windows Presentation Foundation (WPF) apps, Universal Windows Platform (UWP) apps and Windows 10 Mobile

apps. We present XML in more depth, then discuss LINQ to XML, which allows you to query XML content using LINQ syntax.

### *Windows Presentation Foundation (WPF) GUI, Graphics and Multimedia*

Windows Presentation Foundation (WPF)—created after Windows Forms and before UWP—is another Microsoft technology for building robust GUI, graphics and multimedia desktop apps. We discuss WPF in the context of a painting app, a text editor, a color chooser, a book-cover viewer, a television video player, various animations, and speech synthesis and recognition apps.

We featured WPF in the previous edition of this book. Our plans now are to move on to UWP for creating apps that can run on desktop, mobile and other Windows devices. For this reason, the WPF chapters are provided as is from the previous edition—we’ll no longer evolve this material. Many professionals are still actively using Windows Forms and WPF.

### *Case Study: Using the UML to Develop an Object-Oriented Design and C# Implementation of the Software for an ATM (Automated Teller Machine)*

The UML™ (Unified Modeling Language™) is a popular graphical language for visually modeling object-oriented systems. We introduce the UML in the early chapters. We then provide an online object-oriented design case study in which we use the UML to design and implement the software for a simple ATM. We analyze a typical *requirements document* that specifies the details of the system to be built, i.e., *what* the system is supposed to do. We then design the system, specifying *how* it should work—in particular, we

- determine the *classes* needed to implement that system,
- determine the *attributes* the classes need to have,
- determine the *behaviors* the classes’ methods need to exhibit and
- specify how the classes must *interact* with one another to meet the system requirements.

From the design, we then produce a complete working C# implementation. Students in our professional courses often report a “light bulb moment”—the case study helps them “tie it all together” and truly understand object orientation.

## Future Online Bonus Content

Periodically, we *may* make additional bonus chapters and appendices available at

<http://www.informit.com/title/9780134596327>

to registered users of the book. Check this website and/or write to us at [deitel@deitel.com](mailto:deitel@deitel.com) for the status of this content. These *may* cover:

- Universal Windows Platform (UWP) GUI, graphics and multimedia
- ASP.NET web app development
- Web Services
- Microsoft Azure™ Cloud Computing

### *Universal Windows Platform (UWP) for Desktop and Mobile Apps*

The Universal Windows Platform (UWP) is designed to provide a common platform and user experience across all Windows devices, including personal computers, smartphones, tablets, Xbox and even Microsoft’s new HoloLens virtual reality and augmented reality holographic headset—all using nearly identical code.<sup>2</sup>

### *REST Web Services*

Web services enable you to package app functionality in a manner that turns the web into a library of *reusable* services. We used a Flickr REST-based web service in Chapter 21.

### *Microsoft Azure™ Cloud Computing*

Microsoft Azure’s web services enable you to develop, manage and distribute your apps in “the cloud.”

## Notes About the Presentation

*C# 6 for Programmers, 6/e* contains a rich collection of examples. We concentrate on building well-engineered, high performance software and stress program clarity.

*Syntax Shading.* For readability, we syntax shade the code, similar to the way Visual Studio colors the code. Our syntax-shading conventions are:

```

comments appear like this
keywords appear like this
constants and literal values appear like this
all other code appears in black

```

*Code Highlighting.* We emphasize key code segments by placing them in gray rectangles.

*Using Fonts for Emphasis.* We place the key terms and the index’s page reference for each defining occurrence in **bold** text for easy reference. We show on-screen components in the **bold Helvetica** font (for example, the **File** menu) and Visual C# program text in the Lucida font (for example, `int count = 5;`). We use *italics* for emphasis.

*Objectives.* The chapter objectives preview the topics covered in the chapter.

*Programming Tips.* We include programming tips that focus on important aspects of program development. These tips and practices represent the best we’ve gleaned from a combined nine decades of programming, professional training and college teaching experience.



### **Good Programming Practices**

*The Good Programming Practices call attention to techniques that will help you produce programs that are clearer, more understandable and more maintainable.*

2. As of Summer 2016, Windows Forms, WPF and UWP apps all can be posted for distribution, either free or for sale, via the Windows Store. See <http://bit.ly/DesktopToUWP> for more information.



### Common Programming Errors

*Pointing out these Common Programming Errors reduces the likelihood that you'll make them.*



### Error-Prevention Tips

*These tips contain suggestions for exposing and removing bugs from your programs; many of the tips describe aspects of Visual C# that prevent bugs from getting into programs.*



### Performance Tips

*These tips highlight opportunities for making your programs run faster or minimizing the amount of memory that they occupy.*



### Software Engineering Observations

*The Software Engineering Observations highlight architectural and design issues that affect the construction of software systems, especially large-scale systems.*



### Look-and-Feel Observations

*These observations help you design attractive, user-friendly graphical user interfaces that conform to industry norms.*

*Index.* We've included an extensive index for reference. Defining occurrences of key terms in the index are highlighted with a **bold** page number.

## Obtaining the Software Used in *C# 6 for Programmers, 6/e*

We wrote the book's code examples in *C# 6 for Programmers, 6/e* using Microsoft's free Visual Studio 2015 Community edition. See the Before You Begin section that follows this preface for download and installation instructions.

## Microsoft DreamSpark™

Microsoft provides many of its professional developer tools to students for free via a program called DreamSpark (<http://www.dreamspark.com>). If you're a student using this book in a college course, see the website for details on verifying your status so you take advantage of this program.

## Acknowledgments

We'd like to thank Barbara Deitel of Deitel & Associates, Inc. She painstakingly researched the latest versions of Visual C#, Visual Studio, .NET and other key technologies. We'd also like to acknowledge Frank McCown, Ph.D., Associate Professor of Computer Science, Harding University for his suggestion to include an example that used a `ProgressBar` with `async` and `await` in Chapter 21—so we ported to C# a similar example from our book *Java for Programmers, 3/e*.

We're fortunate to have worked with the dedicated team of publishing professionals at Pearson. We appreciate the extraordinary efforts and mentorship of our friend and professional colleague, Mark L. Taub, Editor-in-Chief of the Pearson IT Professional Group.

Kristy Alaura did an extraordinary job recruiting the book's reviewers and managing the review process. Julie Nahil did a wonderful job bringing the book to publication and Chuti Prasertsith worked his magic on the cover design.

## Reviewers

The book was scrutinized by industry C# experts and academics teaching C# courses. They provided countless suggestions for improving the presentation. Any remaining flaws in the book are our own.

**Sixth Edition Reviewers:** Lucian Wischik (Microsoft Visual C# Team), Octavio Hernandez (Microsoft Certified Solutions Developer, Principal Software Engineer at Advanced Bionics), José Antonio González Seco (Parliament of Andalusia, Spain), Bradley Sward (College of Dupage) and Qian Chen (Department of Engineering Technology: Computer Science Technology Program, Savannah State University).

**Other recent edition reviewers:** Douglas B. Bock (MCSD.NET, Southern Illinois University Edwardsville), Dan Crevier (Microsoft), Shay Friedman (Microsoft Visual C# MVP), Amit K. Ghosh (University of Texas at El Paso), Marcelo Guerra Hahn (Microsoft), Kim Hamilton (Software Design Engineer at Microsoft and co-author of *Learning UML 2.0*), Huanhui Hu (Microsoft Corporation), Stephen Hustedde (South Mountain College), James Edward Keysor (Florida Institute of Technology), Narges Kasiri (Oklahoma State University), Helena Kotas (Microsoft), Charles Liu (University of Texas at San Antonio), Chris Lovett (Software Architect at Microsoft), Bashar Lulu (INETA Country Leader, Arabian Gulf), John McIlhinney (Spatial Intelligence; Microsoft MVP Visual Developer, Visual Basic), Ged Mead (Microsoft Visual Basic MVP, DevCity.net), Anand Mukundan (Architect, Polaris Software Lab Ltd.), Dr. Hamid R. Nemati (The University of North Carolina at Greensboro), Timothy Ng (Microsoft), Akira Onishi (Microsoft), Jeffrey P. Scott (Blackhawk Technical College), Joe Stagner (Senior Program Manager, Developer Tools & Platforms, Microsoft), Erick Thompson (Microsoft), Jesús Ubaldo Quevedo-Torrero (University of Wisconsin–Parkside, Department of Computer Science), Shawn Weisfeld (Microsoft MVP and President and Founder of UserGroup.tv) and Zijiang Yang (Western Michigan University).

As you read the book, we'd sincerely appreciate your comments, criticisms, corrections and suggestions for improvement. Please address all correspondence to:

`deitel@deitel.com`

We'll respond promptly. It was fun writing *C# 6 for Programmers, 6/e*—we hope you enjoy reading it!

*Paul Deitel*  
*Harvey Deitel*

## About the Authors

**Paul Deitel**, CEO and Chief Technical Officer of Deitel & Associates, Inc., has over 35 years of experience in computing. He is a graduate of MIT, where he studied Information Technology. Through Deitel & Associates, Inc., he has delivered hundreds of corporate programming training courses worldwide to clients, including Cisco, IBM, Boeing, Siemens, Sun Microsystems (now Oracle), Dell, Fidelity, NASA at the Kennedy Space Cen-

ter, the National Severe Storm Laboratory, NOAA (National Oceanic and Atmospheric Administration), White Sands Missile Range, Rogue Wave Software, SunGard, Nortel Networks, Puma, iRobot, Invensys and many more. He and his co-author, Dr. Harvey Deitel, are the world's best-selling programming-language professional book/textbook/video authors.

Paul was named a Microsoft® Most Valuable Professional (MVP) for C# in 2012–2014. According to Microsoft, “the Microsoft MVP Award is an annual award that recognizes exceptional technology community leaders worldwide who actively share their high quality, real-world expertise with users and Microsoft.” He also holds the Java Certified Programmer and Java Certified Developer designations and is an Oracle Java Champion.



C# MVP 2012–2014

**Dr. Harvey Deitel**, Chairman and Chief Strategy Officer of Deitel & Associates, Inc., has over 55 years of experience in the computer field. Dr. Deitel earned B.S. and M.S. degrees in Electrical Engineering from MIT and a Ph.D. in Mathematics from Boston University—he studied computing in each of these programs before they spun off Computer Science programs. He has extensive college teaching experience, including earning tenure and serving as the Chairman of the Computer Science Department at Boston College before founding Deitel & Associates, Inc., in 1991 with his son, Paul. The Deitels’ publications have earned international recognition, with translations published in Japanese, German, Russian, Spanish, French, Polish, Italian, Simplified Chinese, Traditional Chinese, Korean, Portuguese, Greek, Urdu and Turkish. Dr. Deitel has delivered hundreds of programming courses to corporate, government, military and academic clients.

## About Deitel & Associates, Inc.

Deitel & Associates, Inc., founded by Paul Deitel and Harvey Deitel, is an internationally recognized authoring and corporate training organization, specializing in computer programming languages, object technology, Internet and web software technology, and Android and iOS app development. The company’s clients include many of the world’s largest corporations, government agencies, branches of the military and academic institutions. The company offers instructor-led training courses delivered at client sites worldwide on major programming languages and platforms, including C#®, C++, C, Java™, Android app development, iOS app development, Swift™, Visual Basic® and Internet and web programming.

Through its 40-year publishing partnership with Prentice Hall/Pearson, Deitel & Associates, Inc., creates leading-edge programming professional books, college textbooks, *LiveLessons* video products, e-books and REVEL™ interactive multimedia courses with integrated labs and assessment (<http://revel.pearson.com>). Deitel & Associates, Inc. and the authors can be reached at:

[deitel@deitel.com](mailto:deitel@deitel.com)

To learn more about Deitel’s corporate training curriculum, visit

<http://www.deitel.com/training>

To request a proposal for worldwide on-site, instructor-led training at your organization, send an e-mail to [deitel@deitel.com](mailto:deitel@deitel.com).

Individuals wishing to purchase Deitel books can do so via

<http://bit.ly/DeitelOnAmazon>

Individuals wishing to purchase Deitel *LiveLessons* video training can do so at:

<http://bit.ly/DeitelOnInformit>

Deitel books and *LiveLessons* videos are generally available electronically to Safari Books Online subscribers at:

<http://SafariBooksOnline.com>

You can get a free 10-day Safari Books Online trial at:

<https://www.safaribooksonline.com/register/>

Bulk orders by corporations, the government, the military and academic institutions should be placed directly with Pearson. For more information, visit

<http://www.informit.com/store/sales.aspx>



# Before You Begin

Please read this section before using the book to ensure that your computer is set up properly.

## Font and Naming Conventions

We use fonts to distinguish between features, such as menu names, menu items, and other elements that appear in the program-development environment. Our convention is

- to emphasize Visual Studio features in a **sans-serif bold font** (e.g., **Properties** window) and
- to emphasize program text in a **fixed-width sans-serif font** (e.g., `bool x = true`).

## Visual Studio 2015 Community Edition

This book uses Windows 10 and the free Microsoft Visual Studio 2015 Community edition—Visual Studio also can run on various older Windows versions. Ensure that your system meets Visual Studio 2015 Community edition’s minimum hardware and software requirements listed at:

<https://www.visualstudio.com/en-us/visual-studio-2015-system-requirements-vs>

Next, download the installer from

<https://www.visualstudio.com/products/visual-studio-express-vs>

then execute it and follow the on-screen instructions to install Visual Studio.

Though we developed the book’s examples on Windows 10, most of the examples will run on Windows 7 and higher. Most examples without graphical user interfaces (GUIs) also will run on other C# and .NET implementations—see “If You’re Not Using Microsoft Visual C#...” later in this Before You Begin for more information.

## Viewing File Extensions

Several screenshots in *C# 6 for Programmers, 6/e* display file names with file-name extensions (e.g., `.txt`, `.cs`, `.png`, etc.). You may need to adjust your system’s settings to display file-name extensions. If you’re using Windows 7:

1. Open **Windows Explorer**.
2. Press the **Alt** key to display the menu bar, then select **Folder Options...** from the **Tools** menu.
3. In the dialog that appears, select the **View** tab.

4. In the **Advanced settings** pane, *uncheck* the box to the left of the text **Hide extensions for known file types**.
5. Click **OK** to apply the setting and close the dialog.

If you're using Windows 8 or higher:

1. Open **File Explorer**.
2. Click the **View** tab.
3. Ensure that the **File name extensions** checkbox is *checked*.

## Obtaining the Source Code

*C# 6 for Programmers*, 6/e's source-code examples are available for download at

<http://www.deitel.com/books/CSharp6FP>

Click the **Examples** link to download the ZIP archive file to your computer—most browsers will save the file into your user account's **Downloads** folder. You can extract the ZIP file's contents using built-in Windows capabilities, or using a third-party archive-file tool such as WinZip ([www.winzip.com](http://www.winzip.com)) or 7-zip ([www.7-zip.org](http://www.7-zip.org)).

Throughout the book, steps that require you to access our example code on your computer assume that you've extracted the examples from the ZIP file and placed them in your user account's **Documents** folder. You can extract them anywhere you like, but if you choose a different location, you'll need to update our steps accordingly. To extract the ZIP file's contents using the built-in Windows capabilities:

1. Open **Windows Explorer** (Windows 7) or **File Explorer** (Windows 8 and higher).
2. Locate the ZIP file on your system, typically in your user account's **Downloads** folder.
3. Right click the ZIP file and select **Extract All...**
4. In the dialog that appears, navigate to the folder where you'd like to extract the contents, then click the **Extract** button.

## Configuring Visual Studio for Use with This Book

In this section, you'll use Visual Studio's **Options** dialog to configure several Visual Studio options. Setting these options is not required, but will make your Visual Studio match what we show in the book's Visual Studio screen captures.

### *Visual Studio Theme*

Visual Studio has three color themes—**Blue**, **Dark** and **Light**. We used the **Blue** theme with light colored backgrounds to make the book's screen captures easier to read. To switch themes:

1. In the Visual Studio **Tools** menu, select **Options...** to display the **Options** dialog.
2. In the left column, select **Environment**.
3. Select the **Color theme** you wish to use.

Keep the **Options** dialog open for the next step.

*Line Numbers*

Throughout the book's discussions, we refer to code in our examples by line number. Many programmers find it helpful to display line numbers in Visual Studio as well. To do so:

1. Expand the **Text Editor** node in the **Options** dialog's left pane.
2. Select **All Languages**.
3. In the right pane, check the **Line numbers** checkbox.

Keep the **Options** dialog open for the next step.

*Tab Size for Code Indents*

Microsoft recommends four-space indents in source code, which is the Visual Studio default. Due to the fixed and limited width of code lines in print, we use three-space indents—this reduces the number of code lines that wrap to a new line, making the code a bit easier to read. If you wish to use three-space indents:

1. Expand the **C#** node in the **Options** dialog's left pane and select **Tabs**.
2. Ensure that **Insert spaces** is selected.
3. Enter **3** for both the **Tab size** and **Indent size** fields.
4. Click **OK** to save your settings.

**If You're Not Using Microsoft Visual C#...**

C# can be used on other platforms via two open-source projects managed by the .NET Foundation (<http://www.dotnetfoundation.org>)—the Mono Project and .NET Core.

*Mono Project*

The **Mono Project** is an open source, cross-platform C# and .NET Framework implementation that can be installed on Linux, OS X (soon to be renamed as macOS) and Windows. The code for most of the book's console (non-GUI) apps will compile and run using the Mono Project. Mono also supports Windows Forms GUI, which is used in Chapters 14–15 and several later examples. For more information and to download Mono, visit:

<http://www.mono-project.com/>

*.NET Core*

**.NET Core** is a new cross-platform .NET implementation for Windows, Linux, OS X and FreeBSD. The code for most of the book's console (non-GUI) apps will compile and run using .NET Core. At the time of this writing, a .NET Core version for Windows was available and versions were still under development for other platforms. For more information and to download .NET Core, visit:

<https://dotnet.github.io/>

You're now ready to get started with C# and the .NET platform using *C# 6 for Programmers, 6/e*. We hope you enjoy the book!

# 1

# Introduction

## Objectives

In this chapter you'll:

- Understand the history of the Visual C# programming language and the Windows operating system.
- Learn what cloud computing with Microsoft Azure is.
- Review the basics of object technology.
- Understand the parts that Windows, .NET, Visual Studio and C# play in the C# ecosystem.
- Test-drive a Visual C# drawing app.

- 1.1 Introduction
- 1.2 Object Technology: A Brief Review
- 1.3 C#
  - 1.3.1 Object-Oriented Programming
  - 1.3.2 Event-Driven Programming
  - 1.3.3 Visual Programming
  - 1.3.4 Generic and Functional Programming
  - 1.3.5 An International Standard
  - 1.3.6 C# on Non-Windows Platforms
  - 1.3.7 Internet and Web Programming
  - 1.3.8 Asynchronous Programming with `async` and `await`
- 1.4 Microsoft's .NET
  - 1.4.1 .NET Framework
  - 1.4.2 Common Language Runtime
  - 1.4.3 Platform Independence
  - 1.4.4 Language Interoperability
- 1.5 Microsoft's Windows® Operating System
- 1.6 Visual Studio Integrated Development Environment
- 1.7 Painter Test-Drive in Visual Studio Community

## 1.1 Introduction

Welcome to C#<sup>1</sup>—a powerful computer-programming language that's used to build substantial computer applications. There are billions of personal computers in use and an even larger number of mobile devices with computers at their core. Since it was released in 2001, C# has been used primarily to build applications for personal computers and systems that support them. The explosive growth of mobile phones, tablets and other devices also is creating significant opportunities for programming mobile apps. With this new sixth edition of *C# 6 for Programmers*, you'll be able to use Microsoft's new Universal Windows Platform (UWP) with Windows 10 to build C# apps for both personal computers and Windows 10 Mobile devices. With Microsoft's purchase of Xamarin, you also can develop C# mobile apps for Android devices and for iOS devices, such as iPhones and iPads.

## 1.2 Object Technology: A Brief Review

C# is an object-oriented programming language. In this section we'll review the basics of object technology.

Building software quickly, correctly and economically remains an elusive goal at a time when demands for new and more powerful software are soaring. **Objects**, or more precisely—as we'll see in Chapter 4—the **classes** objects come from, are essentially *reusable* software components. There are date objects, time objects, audio objects, video objects, automobile objects, people objects, etc. Almost any *noun* can be reasonably represented as a software object in terms of *attributes* (e.g., name, color and size) and *behaviors* (e.g., calculating, moving and communicating). Software developers have discovered that using a modular, object-oriented design-and-implementation approach can make software-development groups much more productive than was possible with earlier techniques—object-oriented programs are often easier to understand, correct and modify.

### *The Automobile as an Object*

Let's begin with a simple analogy. Suppose you want to *drive a car and make it go faster by pressing its accelerator pedal*. What must happen before you can do this? Well, before you

---

1. The name C#, pronounced “C-sharp,” is based on the musical # notation for “sharp” notes.

can drive a car, someone has to *design* it. A car typically begins as engineering drawings, similar to the *blueprints* that describe the design of a house. These drawings include the design for an accelerator pedal. The pedal *hides* from the driver the complex mechanisms that actually make the car go faster, just as the brake pedal hides the mechanisms that slow the car, and the steering wheel *hides* the mechanisms that turn the car. This enables people with little or no knowledge of how engines, braking and steering mechanisms work to drive a car easily.

Before you can drive a car, it must be *built* from the engineering drawings that describe it. A completed car has an *actual* accelerator pedal to make the car go faster, but even that's not enough—the car won't accelerate on its own (hopefully!), so the driver must *press* the pedal to accelerate the car.

### *Methods and Classes*

Let's use our car example to introduce some key object-oriented programming concepts. Performing a task in a program requires a **method**. The method houses the program statements that actually perform the task. It *hides* these statements from its user, just as a car's accelerator pedal hides from the driver the mechanisms of making the car go faster. In C#, we create a program unit called a class to house the set of methods that perform the class's tasks. For example, a class that represents a bank account might contain one method to *deposit* money to an account and another to *withdraw* money from an account. A class is similar in concept to a car's engineering drawings, which house the design of an accelerator pedal, steering wheel, and so on.

### *Making Objects from Classes*

Just as someone has to *build a car* from its engineering drawings before you can actually drive a car, you must *build an object* from a class before a program can perform the tasks that the class's methods define. The process of doing this is called *instantiation*. An object is then referred to as an **instance** of its class.

### *Reuse*

Just as a car's engineering drawings can be *reused* many times to build many cars, you can *reuse* a class many times to build many objects. Reuse of existing classes when building new classes and programs saves time and effort. Reuse also helps you build more reliable and effective systems, because existing classes and components often have gone through extensive *testing* (to locate problems), *debugging* (to correct those problems) and *performance tuning*. Just as the notion of *interchangeable parts* was crucial to the Industrial Revolution, reusable classes are crucial to the software revolution that's been spurred by object technology.

### *Messages and Method Calls*

When you drive a car, pressing its gas pedal sends a *message* to the car to perform a task—that is, to go faster. Similarly, you *send messages to an object*. Each message is implemented as a **method call** that tells a method of the object to perform its task. For example, a program might call a particular bank-account object's *deposit* method to increase the account's balance.

### *Attributes and Instance Variables*

A car, besides having capabilities to accomplish tasks, also has *attributes*, such as its color, its number of doors, the amount of gas in its tank, its current speed and its record of total miles driven (i.e., its odometer reading). Like its capabilities, the car's attributes are represented as part of its design in its engineering diagrams (which, for example, include an odometer and a fuel gauge). As you drive an actual car, these attributes are carried along with the car. Every car maintains its *own* attributes. For example, each car knows how much gas is in its own gas tank, but not how much is in the tanks of *other* cars.

An object, similarly, has attributes that it carries along as it's used in a program. These attributes are specified as part of the object's class. For example, a bank-account object has a *balance attribute* that represents the amount of money in the account. Each bank-account object knows the balance in the account it represents, but *not* the balances of the *other* accounts in the bank. Attributes are specified by the class's **instance variables**.

### *Properties, get Accessors and set Accessors*

Attributes are not necessarily accessible directly. The car manufacturer does not want drivers to take apart the car's engine to observe the amount of gas in its tank. Instead, the driver can check the fuel gauge on the dashboard. The bank does not want its customers to walk into the vault to count the amount of money in an account. Instead, the customers talk to a bank teller or check personalized online bank accounts. Similarly, you do not need to have access to an object's instance variables in order to use them. You should use the **properties** of an object. Properties contain **get accessors** for reading the values of variables, and **set accessors** for storing values into them.

### *Encapsulation*

Classes **encapsulate** (i.e., wrap) attributes and methods into objects created from those classes—an object's attributes and methods are intimately related. Objects may communicate with one another, but they're normally not allowed to know how other objects are implemented—implementation details are *hidden* within the objects themselves. This **information hiding**, as we'll see, is crucial to good software engineering.

### *Inheritance*

A new class of objects can be created quickly and conveniently by **inheritance**—the new class absorbs the characteristics of an existing class, possibly customizing them and adding unique characteristics of its own. In our car analogy, an object of class “convertible” certainly *is an* object of the more *general* class “automobile,” but more *specifically*, the roof can be raised or lowered.

### *Object-Oriented Analysis and Design (OOAD)*

Soon you'll be writing programs in C#. How will you create the code for your programs? Perhaps, like many programmers, you'll simply turn on your computer and start typing. This approach may work for small programs (like the ones we present in the early chapters of the book), but what if you were asked to create a software system to control thousands of automated teller machines for a major bank? Or suppose you were asked to work on a team of thousands of software developers building the next generation of the U.S. air traffic control system? For projects so large and complex, you should not simply sit down and start writing programs.

To create the best solutions, you should follow a detailed **analysis** process for determining your project's **requirements** (i.e., defining *what* the system is supposed to do) and developing a **design** that satisfies them (i.e., deciding *how* the system should do it). Ideally, you'd go through this process and carefully review the design (and have your design reviewed by other software professionals) before writing any code. If this process involves analyzing and designing your system from an object-oriented point of view, it's called an **object-oriented analysis and design (OOAD) process**. Languages like C# are object oriented—programming in such a language, called **object-oriented programming (OOP)**, allows you to implement an object-oriented design as a working system.

### *The UML (Unified Modeling Language)*

Although many different OOAD processes exist, a single graphical language for communicating the results of *any* OOAD process has come into wide use. This language, known as the Unified Modeling Language (UML), is now the most widely used graphical scheme for modeling object-oriented systems. We present our first UML diagrams in Chapters 4 and 5, then use them in our deeper treatment of object-oriented programming through Chapter 12. In our online ATM Software Engineering Case Study, we present a simple subset of the UML's features as we guide you through an object-oriented design and implementation experience.

## 1.3 C#

In 2000, Microsoft announced the **C#** programming language. C# has roots in the C, C++ and Java programming languages. It has similar capabilities to Java and is appropriate for the most demanding app-development tasks, especially for building today's desktop apps, large-scale enterprise apps, and web-based, mobile and cloud-based apps.

### 1.3.1 Object-Oriented Programming

C# is *object oriented*—we've discussed the basics of object technology and we present a rich treatment of object-oriented programming throughout the book. C# has access to the powerful **.NET Framework Class Library**—a vast collection of prebuilt classes that enable you to develop apps quickly (Fig. 1.1). We'll say more about .NET in Section 1.4.

Some key capabilities in the .NET Framework Class Library	
Database	Debugging
Building web apps	Multithreading
Graphics	File processing
Input/output	Security
Computer networking	Web communication
Permissions	Graphical user interface
Mobile	Data structures
String processing	Universal Windows Platform GUI

**Fig. 1.1** | Some key capabilities in the .NET Framework Class Library.



### 1.3.2 Event-Driven Programming

C# graphical user interfaces (GUIs) are **event driven**. You can write programs that respond to user-initiated **events** such as mouse clicks, keystrokes, timer expirations and *touches* and *finger swipes*—gestures that are widely used on smartphones and tablets.

### 1.3.3 Visual Programming

Visual Studio enables you to use C# as a *visual programming language*—in addition to writing program statements to build portions of your apps, you’ll also use Visual Studio to drag and drop predefined GUI objects like *buttons* and *textboxes* into place on your screen, and label and resize them. Visual Studio will write much of the GUI code for you.

### 1.3.4 Generic and Functional Programming

#### *Generic Programming*

It’s common to write a program that processes a collection—e.g., a collection of numbers, a collection of contacts, a collection of videos, etc. Historically, you had to program separately to handle each type of collection. With generic programming, you write code that handles a collection “in the general” and C# handles the specifics for each collection type, saving you a great deal of work. Chapters 18–19 present generics and generic collections.

#### *Functional Programming*

With *functional programming*, you specify *what* you want to accomplish in a task, but *not how* to accomplish it. For example, with Microsoft’s LINQ—which we introduce in Chapter 9, then use in many later chapters—you can say, “Here’s a collection of numbers, give me the sum of its elements.” You do *not* need to specify the mechanics of walking through the elements and adding them into a running total one at a time—LINQ handles all that for you. Functional programming speeds application development and reduces errors. We take a deeper look at functional programming in Chapter 19.

### 1.3.5 An International Standard

C# has been standardized through ECMA International:

<http://www.ecma-international.org>

This enables other implementations of the language besides Microsoft’s Visual C#. At the time of this writing, the C# standard document—ECMA-334—was still being updated for C# 6. For information on ECMA-334, visit

<http://www.ecma-international.org/publications/standards/Ecma-334.htm>

Visit the Microsoft download center to find the latest version of Microsoft’s C# 6 specification, other documentation and software downloads.

### 1.3.6 C# on Non-Windows Platforms

Microsoft originally developed C# for Windows development, but it can be used on other platforms via the **Mono Project** and **.NET Core**—both managed by the .NET Foundation

<http://www.dotnetfoundation.org/>

For more information, see the Before You Begin section after the Preface.

### 1.3.7 Internet and Web Programming

Today's apps can be written with the aim of communicating among the world's computers. As you'll see, this is the focus of Microsoft's .NET strategy. Later in the book, you'll build web-based apps with C# and Microsoft's **ASP.NET** technology.

### 1.3.8 Asynchronous Programming with `async` and `await`

In most programming today, each task in a program must finish executing before the next task can begin. This is called *synchronous programming* and is the style we use for most of this book. C# also allows *asynchronous programming* in which multiple tasks can be performed at the *same* time. Asynchronous programming can help you make your apps more responsive to user interactions, such as mouse clicks and keystrokes, among many other uses.

Asynchronous programming in early versions of Visual C# was difficult and error prone. C#'s `async` and `await` capabilities simplify asynchronous programming by enabling the compiler to hide much of the associated complexity from the developer. In Chapter 21, we provide an introduction to asynchronous programming with `async` and `await`.

## 1.4 Microsoft's .NET

In 2000, Microsoft announced its **.NET initiative** ([www.microsoft.com/net](http://www.microsoft.com/net)), a broad vision for using the Internet and the web in the development, engineering, distribution and use of software. Rather than forcing you to use a single programming language, .NET permits you to create apps in *any* .NET-compatible language (such as C#, Visual Basic, Visual C++ and many others). Part of the initiative includes Microsoft's ASP.NET technology for building web-based applications.

### 1.4.1 .NET Framework

The **.NET Framework Class Library** provides many capabilities that you'll use to build substantial C# apps quickly and easily. It contains *thousands* of valuable *prebuilt* classes that have been tested and tuned to maximize performance. You'll learn how to create your own classes, but you should *re-use* the .NET Framework classes whenever possible to speed up the software-development process, while enhancing the quality and performance of the software you develop.

### 1.4.2 Common Language Runtime

The **Common Language Runtime (CLR)**, another key part of the .NET Framework, executes .NET programs and provides functionality to make them easier to develop and debug. The CLR is a **virtual machine (VM)**—software that manages the execution of programs and hides from them the underlying operating system and hardware. The source code for programs that are executed and managed by the CLR is called *managed code*. The CLR provides various services to managed code, such as

- integrating software components written in different .NET languages,
- error handling between such components,
- enhanced security,
- automatic memory management and more.

Unmanaged-code programs do not have access to the CLR's services, which makes unmanaged code more difficult to write.<sup>2</sup> Managed code is compiled into machine-specific instructions in the following steps:

1. First, the code is compiled into **Microsoft Intermediate Language (MSIL)**. Code converted into MSIL from other languages and sources can be woven together by the CLR—this allows programmers to work in their preferred .NET programming language. The MSIL for an app's components is placed into the app's *executable file*—the file that causes the computer to perform the app's tasks.
2. When the app executes, another compiler (known as the **just-in-time compiler** or **JIT compiler**) in the CLR translates the MSIL in the executable file into machine-language code (for a particular platform).
3. The machine-language code executes on that platform.

### 1.4.3 Platform Independence

If the .NET Framework exists and is installed for a platform, that platform can run *any* .NET program. The ability of a program to run without modification across multiple platforms is known as **platform independence**. Code written once can be used on another type of computer without modification, saving time and money. In addition, software can target a wider audience. Previously, companies had to decide whether converting their programs to different platforms—a process called **porting**—was worth the cost. With .NET, porting programs is no longer an issue, at least once .NET itself has been made available on the platforms.

### 1.4.4 Language Interoperability

The .NET Framework provides a high level of **language interoperability**. Because software components written in different .NET languages (such as C# and Visual Basic) are all compiled into MSIL, the components can be combined to create a single unified program. Thus, MSIL allows the .NET Framework to be **language independent**.

The .NET Framework Class Library can be used by any .NET language. The latest release of .NET includes .NET 4.6 and .NET Core:

- .NET 4.6 introduces many improvements and new features, including ASP.NET 5 for web-based applications, improved support for today's high-resolution 4K screens and more.
- .NET Core is the cross-platform subset of .NET for Windows, Linux, OS X and FreeBSD.

## 1.5 Microsoft's Windows® Operating System

Microsoft's Windows is the most widely personal-computer, desktop operating system worldwide. **Operating systems** are software systems that make using computers more convenient for users, developers and system administrators. They provide *services* that allow each app to execute safely, efficiently and *concurrently* (i.e., in parallel) with other apps.

---

2. <http://msdn.microsoft.com/library/8bs2ecf4>.

Other popular desktop operating systems include macOS (formerly OS X) and Linux. *Mobile operating systems* used in smartphones and tablets include Microsoft's Windows 10 Mobile, Google's Android and Apple's iOS (for iPhone, iPad and iPod Touch devices). Figure 1.2 presents the evolution of the Windows operating system.

Version	Description
Windows in the 1990s	In the mid-1980s, Microsoft developed the <b>Windows operating system</b> based on a graphical user interface with buttons, textboxes, menus and other graphical elements. The various versions released throughout the 1990s were intended for personal computing. Microsoft entered the corporate operating systems market with the 1993 release of <i>Windows NT</i> .
Windows XP and Windows Vista	<i>Windows XP</i> was released in 2001 and combined Microsoft's corporate and consumer operating-system lines. At the time of this writing, it still holds more than 10% of the operating-systems market ( <a href="https://www.netmarketshare.com/operating-system-market-share.aspx">https://www.netmarketshare.com/operating-system-market-share.aspx</a> ). <i>Windows Vista</i> , released in 2007, offered the attractive new Aero user interface, many powerful enhancements and new apps and enhanced security. But Vista never caught on.
Windows 7	<i>Windows 7</i> is currently the world's most widely used desktop operating system with over 47% of the operating-systems market ( <a href="https://www.netmarketshare.com/operating-system-market-share.aspx">https://www.netmarketshare.com/operating-system-market-share.aspx</a> ). Windows added enhancements to the Aero user interface, faster startup times, further refinement of Vista's security features, touch-screen with multitouch support, and more.
Windows 8 for Desktops and Tablets	Windows 8, released in 2012, provided a similar platform (the underlying system on which apps run) and <i>user experience</i> across a wide range of devices including personal computers, smartphones, tablets <i>and</i> the Xbox Live online game service. Its new look-and-feel featured a Start screen with <i>tiles</i> representing each app, similar to that of <i>Windows Phone</i> (now Windows 10 Mobile)—Microsoft's smartphone operating system. Windows 8 featured <i>multi-touch</i> support for <i>touchpads</i> and <i>touchscreen</i> devices, enhanced security features and more.
Windows 8 UI (User Interface)	Windows 8 UI (previously called "Metro") introduced a clean look-and-feel with minimal distractions to the user. Windows 8 apps featured a <i>chromeless window</i> with no borders, title bars and menus. These elements were <i>hidden</i> , allowing apps to fill the <i>entire</i> screen—particularly helpful on smaller screens such as tablets and smartphones. The interface elements were displayed in the <i>app bar</i> when the user <i>swiped</i> the top or bottom of the screen by holding down the mouse button, moving the mouse in the swipe direction and releasing the mouse button; or using a <i>finger swipe</i> on a touch-screen device.

**Fig. 1.2** | The evolution of the Windows operating system. (Part 1 of 2.)

Version	Description
Windows 10 and the Universal Windows Platform	Windows 10, released in 2015, is the current version of Windows and currently holds a 15% (and growing) share of the operating-systems market ( <a href="https://www.netmarketshare.com/operating-system-market-share.aspx">https://www.netmarketshare.com/operating-system-market-share.aspx</a> ). In addition to many user-interface and other updates, Windows 10 introduced the <b>Universal Windows Platform (UWP)</b> , which is designed to provide a common platform (the underlying system on which apps run) and user experience across all Windows devices including personal computers, smartphones, tablets, Xbox and even Microsoft's new HoloLens augmented reality holographic headset—all using nearly identical code.

**Fig. 1.2** | The evolution of the Windows operating system. (Part 2 of 2.)

### Windows Store

You can sell apps or offer them for free in the Windows Store. At the time of this writing, the fee to become a registered developer is \$19 for individuals and \$99 for companies. Microsoft retains 30% of the purchase price (more in some markets). See the App Developer Agreement for more information:

<https://msdn.microsoft.com/en-us/library/windows/apps/hh694058.aspx>

The Windows Store offers several business models for monetizing your app. You can charge full price for your app before download, with prices starting at \$1.49. You also can offer a time-limited trial or feature-limited trial that allows users to try the app before purchasing the full version, sell virtual goods (such as additional app features) using in-app purchases and more. To learn more about the Windows Store and monetizing your apps, visit

<https://msdn.microsoft.com/windows/uwp/monetize/index>

## 1.6 Visual Studio Integrated Development Environment

C# programs can be created using Microsoft's Visual Studio—a collection of software tools called an **Integrated Development Environment (IDE)**. The **Visual Studio Community** edition IDE enables you to *write, run, test* and *debug* C# programs quickly and conveniently. It also supports Microsoft's Visual Basic, Visual C++ and F# programming languages and many more. Most of this book's examples were built using *Visual Studio Community*, which runs on Windows 7, 8 and 10. A few of the book's examples require Windows 10.

## 1.7 Painter Test-Drive in Visual Studio Community

You'll now use *Visual Studio Community* to “test-drive” an existing app that enables you to draw on the screen using the mouse. The **Painter** app allows you to choose among several brush sizes and colors. The elements and functionality you see in this app are typical of what you'll learn to program in this text. The following steps walk you through test-driving the app. For this test drive, we assume that you placed the book's examples in your user account's Documents folder in a subfolder named `examples`.

**Step 1: Checking Your Setup**

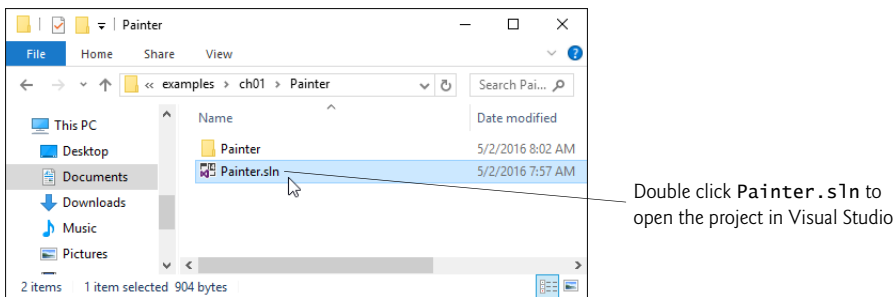
Confirm that you've set up your computer and the software properly by reading the book's Before You Begin section that follows the Preface.

**Step 2: Locating the Painter App's Directory**

Open a File Explorer (Windows 8 and 10) or Windows Explorer (Windows 7) window and navigate to

```
C:\Users\yourUserName\Documents\examples\ch01
```

Double click the Painter folder to view its contents (Fig. 1.3), then double click the Painter.sln file to open the app's solution in Visual Studio. An app's *solution* contains all of the app's *code files*, *supporting files* (such as *images*, *videos*, *data files*, etc.) and configuration information. We'll discuss the contents of a solution in more detail in the next chapter.



**Fig. 1.3** | Contents of C:\examples\ch01\Painter.

Depending on your system configuration, File Explorer or Windows Explorer might display `Painter.sln` simply as `Painter`, without the filename extension `.sln`. To display the filename extensions in Windows 8 and higher:

1. Open File Explorer.
2. Click the **View** tab, then ensure that the **File name extensions** checkbox is checked.

To display them in Windows 7:

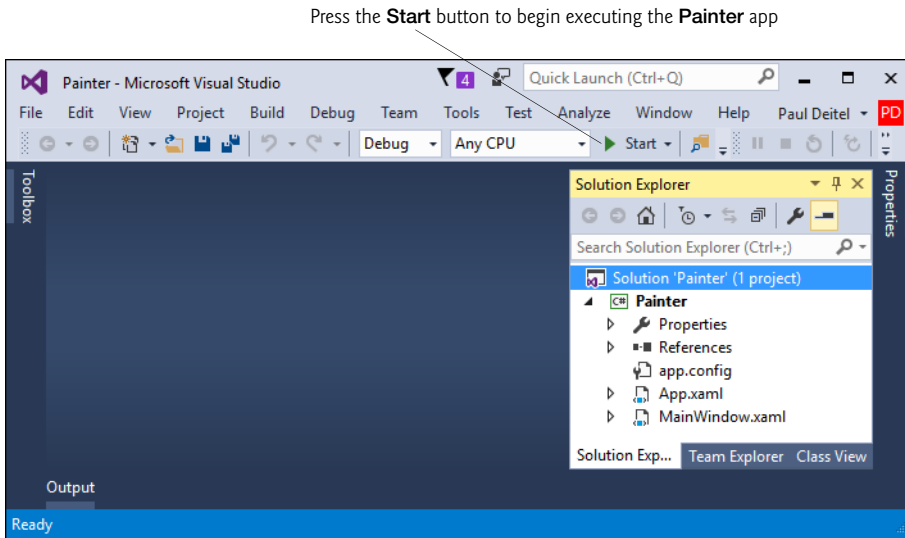
1. Open Windows Explorer.
2. Press *Alt* to display the menu bar, then select **Folder Options...** from Windows Explorer's **Tools** menu.
3. In the dialog that appears, select the **View** tab.
4. In the **Advanced settings:** pane, uncheck the box to the left of the text **Hide extensions for known file types**. [*Note:* If this item is already unchecked, no action needs to be taken.]
5. Click **OK** to apply the setting and close the dialog.

**Step 3: Running the Painter App**

To see the running **Painter** app, click the **Start** button (Fig. 1.4)



or press the *F5* key.



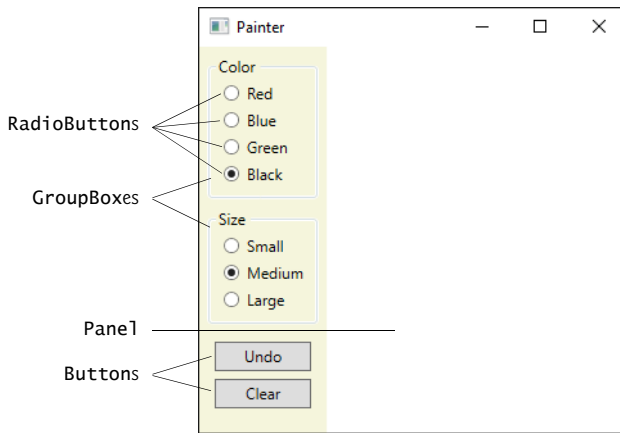
**Fig. 1.4** | Running the **Painter** app.

Figure 1.5 shows the running app and labels several of the app’s graphical elements—called **controls**. These include **GroupBoxes**, **RadioButton**s, **Buttons** and a **Pane1**. These controls and many others are discussed throughout the text. The app allows you to draw with a **Red**, **Blue**, **Green** or **Black** brush of **Small**, **Medium** or **Large** size. As you drag the mouse on the white **Pane1**, the app draws circles of the specified color and size at the mouse pointer’s current position. The slower you drag the mouse, the closer the circles will be. Thus, dragging slowly draws a continuous line (as in Fig. 1.6) and dragging quickly draws individual circles with space in between. You also can **Undo** your previous operation or **Clear** the drawing to start from scratch by pressing the **Buttons** below the **RadioButtons** in the GUI. By using existing *controls*—which are *objects*—you can create powerful apps much faster than if you had to write all the code yourself. This is a key benefit of *software reuse*.

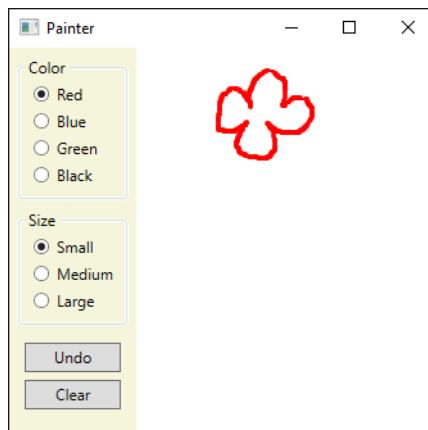
The brush’s properties, selected in the **RadioButtons** labeled **Black** and **Medium**, are *default settings*—the initial settings you see when you first run the app. Programmers include default settings to provide *reasonable* choices that the app will use if the user *does not* change the settings. Default settings also provide visual cues for users to choose their own settings. Now you’ll choose your own settings as a user of this app.

**Step 4: Changing the Brush Color**

Click the **RadioButton** labeled **Red** to change the brush color, then click the **RadioButton** labeled **Small** to change the brush size. Position the mouse over the white **Pane1**, then drag the mouse to draw with the brush. Draw flower petals, as shown in Fig. 1.6.



**Fig. 1.5** | Painter app running in Windows 10.



**Fig. 1.6** | Drawing flower petals with a small red brush.

#### *Step 5: Changing the Brush Color and Size*

Click the **Green** **RadioButton** to change the brush color. Then, click the **Large** **RadioButton** to change the brush size. Draw grass and a flower stem, as shown in Fig. 1.7.

#### *Step 6: Finishing the Drawing*

Click the **Blue** and **Medium** **RadioButton**s. Draw raindrops, as shown in Fig. 1.8, to complete the drawing.

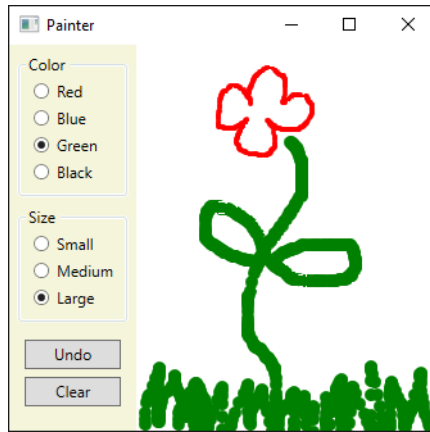
#### *Step 7: Stopping the App*

When you run an app from Visual Studio, you can terminate it by clicking the stop button

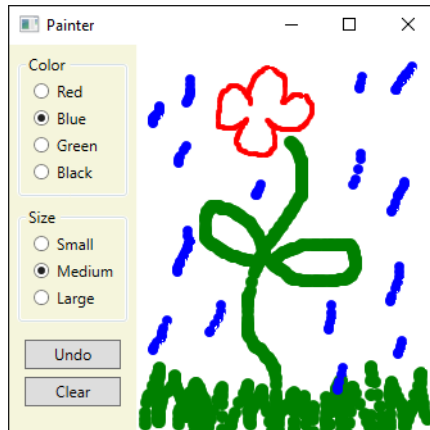


on the Visual Studio toolbar or by clicking the close box





**Fig. 1.7** | Drawing the flower stem and grass with a large green brush.



**Fig. 1.8** | Drawing rain drops with a medium blue brush.



on the running app's window.

Now that you've completed the test-drive, you're ready to begin developing C# apps. In Chapter 2, Introduction to Visual Studio and Visual Programming, you'll use Visual Studio to create your first C# program using *visual programming* techniques. As you'll see, Visual Studio will generate for you the code that builds the app's GUI. In Chapter 3, Introduction to C# App Programming, you'll begin writing C# programs containing conventional program code that you write.

# 7

## Methods: A Deeper Look

### Objectives

In this chapter you'll:

- See that `static` methods and variables are associated with classes rather than objects.
- Use common `Math` class functions.
- Learn C#'s argument promotion rules for when argument types do not match parameter types exactly.
- Get a high-level overview of various namespaces from the .NET Framework Class Library.
- Use random-number generation to implement game-playing apps.
- Understand how the visibility of identifiers is limited to specific regions of programs.
- See how the method call and return mechanism is supported by the method-call stack.
- Create overloaded methods.
- Use optional and named parameters.
- Use recursive methods.
- Understand what value types and reference types are.
- Pass method arguments by value and by reference.

- 7.1 Introduction
- 7.2 Packaging Code in C#
- 7.3 **static** Methods, **static** Variables and Class **Math**
  - 7.3.1 **Math** Class Methods
  - 7.3.2 **Math** Class Constants **PI** and **E**
  - 7.3.3 Why Is **Main** Declared **static**?
  - 7.3.4 Additional Comments About **Main**
- 7.4 Methods with Multiple Parameters
  - 7.4.1 Keyword **static**
  - 7.4.2 Method **Maximum**
  - 7.4.3 Assembling **strings** with Concatenation
  - 7.4.4 Breaking Apart Large **string** Literals
  - 7.4.5 When to Declare Variables as Fields
  - 7.4.6 Implementing Method **Maximum** by Reusing Method **Math.Max**
- 7.5 Notes on Using Methods
- 7.6 Argument Promotion and Casting
  - 7.6.1 Promotion Rules
  - 7.6.2 Sometimes Explicit Casts Are Required
- 7.7 The .NET Framework Class Library
- 7.8 Case Study: Random-Number Generation
  - 7.8.1 Creating an Object of Type **Random**
  - 7.8.2 Generating a Random Integer
  - 7.8.3 Scaling the Random-Number Range
  - 7.8.4 Shifting Random-Number Range
  - 7.8.5 Combining Shifting and Scaling
  - 7.8.6 Rolling a Six-Sided Die
  - 7.8.7 Scaling and Shifting Random Numbers
  - 7.8.8 Repeatability for Testing and Debugging
- 7.9 Case Study: A Game of Chance; Introducing Enumerations
  - 7.9.1 Method **RollDice**
  - 7.9.2 Method **Main**'s Local Variables
  - 7.9.3 **enum** Type **Status**
  - 7.9.4 The First Roll
  - 7.9.5 **enum** Type **DiceNames**
  - 7.9.6 Underlying Type of an **enum**
  - 7.9.7 Comparing Integers and **enum** Constants
- 7.10 Scope of Declarations
- 7.11 Method-Call Stack and Activation Records
  - 7.11.1 Method-Call Stack
  - 7.11.2 Stack Frames
  - 7.11.3 Local Variables and Stack Frames
  - 7.11.4 Stack Overflow
  - 7.11.5 Method-Call Stack in Action
- 7.12 Method Overloading
  - 7.12.1 Declaring Overloaded Methods
  - 7.12.2 Distinguishing Between Overloaded Methods
  - 7.12.3 Return Types of Overloaded Methods
- 7.13 Optional Parameters
- 7.14 Named Parameters
- 7.15 C# 6 Expression-Bodied Methods and Properties
- 7.16 Recursion
  - 7.16.1 Base Cases and Recursive Calls
  - 7.16.2 Recursive Factorial Calculations
  - 7.16.3 Implementing Factorial Recursively
- 7.17 Value Types vs. Reference Types
- 7.18 Passing Arguments By Value and By Reference
  - 7.18.1 **ref** and **out** Parameters
  - 7.18.2 Demonstrating **ref**, **out** and Value Parameters
- 7.19 Wrap-Up

## 7.1 Introduction

In this chapter, we take a deeper look at methods. We'll discuss the difference between non-**static** and **static** methods. You'll see that the **Math** class in the .NET Framework Class Library provides many **static** methods to perform mathematical calculations. We'll also discuss **static** variables (known as class variables) and why method **Main** is declared **static**.

You'll declare a method with multiple parameters and use operator **+** to perform **string** concatenations. We'll discuss C#'s argument promotion rules for implicitly con-

verting simple-type values to other types and when these rules are applied by the compiler. We'll also present several commonly used Framework Class Library namespaces.

We'll take a brief, and hopefully entertaining, diversion into simulation techniques with random-number generation and develop a version of a popular casino dice game that uses most of the programming techniques you've learned so far. You'll declare named constants with the `const` keyword and with `enum` types. We'll then present C#'s scope rules, which determine where identifiers can be referenced in an app.

We'll discuss how the method-call stack enables C# to keep track of which method is currently executing, how local variables of methods are maintained in memory and how a method knows where to return after it completes execution. You'll overload methods in a class by providing methods with the same name but different numbers and/or types of parameters, and learn how to use optional and named parameters.

We'll introduce C#'s expression-bodied methods, which provide a concise notation for methods that simply return a value to their caller. We'll also use this expression-bodied notation for a read-only property's `get` accessor.

We'll discuss how recursive methods call themselves, breaking larger problems into smaller subproblems until eventually the original problem is solved. Finally, we'll provide more insight into how value-type and reference-type arguments are passed to methods.

## 7.2 Packaging Code in C#

So far, we've used properties, methods and classes to package code. We'll present additional packaging mechanisms in later chapters. C# apps are written by combining your properties, methods and classes with predefined properties, methods and classes available in the .NET Framework Class Library and in other class libraries. Related classes are often grouped into namespaces and compiled into class libraries so that they can be reused in other apps. You'll learn how to create your own namespaces and class libraries in Chapter 15. The Framework Class Library provides many *predefined* classes that contain methods for performing common mathematical calculations, string manipulations, character manipulations, input/output operations, graphical user interfaces, graphics, multimedia, printing, file processing, database operations, networking operations, error checking, web-app development, accessibility (for people with disabilities) and more.



### Software Engineering Observation 7.1

*Don't try to "reinvent the wheel." When possible, reuse Framework Class Library classes and methods (<https://msdn.microsoft.com/library/mt472912>). This reduces app development time and errors, contributes to good performance and often enhances security.*

## 7.3 static Methods, static Variables and Class Math

Although most methods are called to operate on the data of specific objects, this is not always the case. Sometimes a method performs a task that does *not* depend on the data of any object (other than the method's arguments). Such a method applies to the class in which it's declared as a whole and is known as a static method.

It's common for a class to contain a group of static methods to perform common tasks. For example, recall that we used static method `Pow` of class `Math` to raise a value to a power in Fig. 6.6. To declare a method as static, place the keyword `static` before the return type in the method's declaration. You call any static method by specifying the name of the class in which the method is declared, followed by the member-access operator (`.`) and the method name, as in

```
ClassName.MethodName(arguments)
```

### 7.3.1 Math Class Methods

Class `Math` (from the `System` namespace) provides a collection of static methods that enable you to perform common mathematical calculations. For example, you can calculate the square root of `900.0` with the static method call

```
double value = Math.Sqrt(900.0);
```

The expression `Math.Sqrt(900.0)` evaluates to `30.0`. Method `Sqrt` takes an argument of type `double` and returns a result of type `double`. The following statement displays in the console window the value of the preceding method call:

```
Console.WriteLine(Math.Sqrt(900.0));
```

Here, the value that `Sqrt` returns becomes the argument to `WriteLine`. We did not create a `Math` object before calling `Sqrt`, nor did we create a `Console` object before calling `WriteLine`. Also, *all* of `Math`'s methods are static—therefore, each is called by preceding the name of the method with the class name `Math` and the member-access operator (`.`).

Method arguments may be constants, variables or expressions. If `c = 13.0`, `d = 3.0` and `f = 4.0`, then the statement

```
Console.WriteLine(Math.Sqrt(c + d * f));
```

calculates and displays the square root of  $13.0 + 3.0 * 4.0 = 25.0$ —namely, `5.0`. Figure 7.1 summarizes several `Math` class methods. In the figure, `x` and `y` are of type `double`.

Method	Description	Example
<code>Abs(x)</code>	absolute value of <code>x</code>	<code>Abs(23.7)</code> is 23.7 <code>Abs(0.0)</code> is 0.0 <code>Abs(-23.7)</code> is 23.7
<code>Ceiling(x)</code>	rounds <code>x</code> to the smallest integer not less than <code>x</code>	<code>Ceiling(9.2)</code> is 10.0 <code>Ceiling(-9.8)</code> is -9.0
<code>Floor(x)</code>	rounds <code>x</code> to the largest integer not greater than <code>x</code>	<code>Floor(9.2)</code> is 9.0 <code>Floor(-9.8)</code> is -10.0
<code>Cos(x)</code>	trigonometric cosine of <code>x</code> ( <code>x</code> in radians)	<code>Cos(0.0)</code> is 1.0
<code>Sin(x)</code>	trigonometric sine of <code>x</code> ( <code>x</code> in radians)	<code>Sin(0.0)</code> is 0.0
<code>Tan(x)</code>	trigonometric tangent of <code>x</code> ( <code>x</code> in radians)	<code>Tan(0.0)</code> is 0.0

**Fig. 7.1** | Math class methods. (Part 1 of 2.)

Method	Description	Example
Exp( $x$ )	exponential method $e^x$	Exp(1.0) is 2.71828 Exp(2.0) is 7.38906
Log( $x$ )	natural logarithm of $x$ (base $e$ )	Log(Math.E) is 1.0 Log(Math.E * Math.E) is 2.0
Max( $x$ , $y$ )	larger value of $x$ and $y$	Max(2.3, 12.7) is 12.7 Max(-2.3, -12.7) is -2.3
Min( $x$ , $y$ )	smaller value of $x$ and $y$	Min(2.3, 12.7) is 2.3 Min(-2.3, -12.7) is -12.7
Pow( $x$ , $y$ )	$x$ raised to the power $y$ (i.e., $x^y$ )	Pow(2.0, 7.0) is 128.0 Pow(9.0, 0.5) is 3.0
Sqrt( $x$ )	square root of $x$	Sqrt(900.0) is 30.0

**Fig. 7.1** | Math class methods. (Part 2 of 2.)

### 7.3.2 Math Class Constants PI and E

Each object of a class maintains its own copy of each of the class's *instance variables*. There are also variables for which each object of a class does *not* need its own separate copy (as you'll see momentarily). Such variables are declared `static` and are also known as **class variables**. When objects of a class containing `static` variables are created, all the objects of that class share *one* copy of those variables. Together a class's `static` variables and instance variables are known as its **fields**. You'll learn more about `static` fields in Section 10.9.

Class `Math` also declares two `double` constants for commonly used mathematical values:

- **Math.PI** (3.1415926535897931) is the ratio of a circle's circumference to its diameter, and
- **Math.E** (2.7182818284590451) is the base value for natural logarithms (calculated with `static` `Math` method `Log`).

These constants are declared in class `Math` with the modifiers `public` and `const`. Making them `public` allows other programmers to use these variables in their own classes. A constant is declared with the keyword `const`—its value cannot be changed after the constant is declared. Fields declared `const` are implicitly `static`, so you can access them via the class name `Math` and the member-access operator (`.`), as in `Math.PI` and `Math.E`.



#### Common Programming Error 7.1

*Constants declared in a class, but not inside a method or property, are implicitly `static`—it's a syntax error to declare such a constant with keyword `static` explicitly.*

### 7.3.3 Why Is `Main` Declared `static`?

Why must `Main` be declared `static`? During app *startup*, when *no* objects of the class have been created, the `Main` method must be called to begin program execution. `Main` is sometimes called the app's **entry point**. Declaring `Main` as `static` allows the execution environment to invoke `Main` without creating an instance of the class. Method `Main` is typically declared with the header:

```
static void Main()
```

but also can be declared with the header:

```
static void Main(string[] args)
```

which we'll discuss and demonstrate in Section 8.12, Shuffling and Dealing Cards. In addition, you can declare `Main` with return type `int` (instead of `void`)—this can be useful if an app is executed by another app and needs to return an indication of success or failure to that other app.

### 7.3.4 Additional Comments About Main

Most earlier examples have one class that contained only `Main`, and some examples had a second class that was used by `Main` to create and manipulate objects. Actually, *any* class can contain a `Main` method. In fact, each of our two-class examples could have been implemented as one class. For example, in the app in Figs. 4.11–4.12, method `Main` (lines 7–43 of Fig. 4.12) could have been moved into class `Account` (Fig. 4.11). The app results would have been identical to those of the two-class version. You can place a `Main` method in every class you declare. Some programmers take advantage of this to build a small test app into each class they declare. However, if you declare more than one `Main` method among the classes of your project, you'll need to indicate to the IDE which one you would like to be the *app's* entry point. To do so:

1. With the project open in Visual Studio, select **Project > [ProjectName] Properties...** (where `[ProjectName]` is the name of your project).
2. Select the class containing the `Main` method that should be the entry point from the **Startup object** list box.

## 7.4 Methods with Multiple Parameters

We now consider how to write a method with multiple parameters. Figure 7.2 defines `Maximum` method that determines and returns the largest of *three* `double` values. When the app begins execution, the `Main` method (lines 8–23) executes. Line 19 calls method `Maximum` (declared in lines 26–43) to determine and return the largest of its three `double` arguments. In Section 7.4.3, we'll discuss the use of the `+` operator in line 22. The sample outputs show that `Maximum` determines the largest value regardless of whether that value is the first, second or third argument.

---

```

1 // Fig. 7.2: MaximumFinder.cs
2 // Method Maximum with three parameters.
3 using System;
4
5 class MaximumFinder
6 {
7     // obtain three floating-point values and determine maximum value
8     static void Main()
9     {
```

---

**Fig. 7.2** | Method `Maximum` with three parameters. (Part 1 of 2.)

```
10 // prompt for and input three floating-point values
11 Console.WriteLine("Enter first floating-point value: ");
12 double number1 = double.Parse(Console.ReadLine());
13 Console.WriteLine("Enter second floating-point value: ");
14 double number2 = double.Parse(Console.ReadLine());
15 Console.WriteLine("Enter third floating-point value: ");
16 double number3 = double.Parse(Console.ReadLine());
17
18 // determine the maximum of three values
19 double result = Maximum(number1, number2, number3);
20
21 // display maximum value
22 Console.WriteLine("Maximum is: " + result);
23 }
24
25 // returns the maximum of its three double parameters
26 static double Maximum(double x, double y, double z)
27 {
28     double maximumValue = x; // assume x is the largest to start
29
30     // determine whether y is greater than maximumValue
31     if (y > maximumValue)
32     {
33         maximumValue = y;
34     }
35
36     // determine whether z is greater than maximumValue
37     if (z > maximumValue)
38     {
39         maximumValue = z;
40     }
41
42     return maximumValue;
43 }
44 }
```

```
Enter first floating-point values: 3.33
Enter second floating-point values: 1.11
Enter third floating-point values: 2.22
Maximum is: 3.33
```

```
Enter first floating-point values: 2.22
Enter second floating-point values: 3.33
Enter third floating-point values: 1.11
Maximum is: 3.33
```

```
Enter first floating-point values: 2.22
Enter second floating-point values: 1.11
Enter third floating-point values: 3.33
Maximum is: 3.33
```

**Fig. 7.2** | Method Maximum with three parameters. (Part 2 of 2.)



### 7.4.1 Keyword `static`

Method `Maximum`'s declaration begins with keyword `static`, which enables the `Main` method (another `static` method) to call `Maximum` as shown in line 19 without creating an object of class `MaximumFinder` and without qualifying the method name with the class name `MaximumFinder`—`static` methods in the *same* class can call each other directly.

### 7.4.2 Method `Maximum`

Consider the declaration of method `Maximum` (lines 26–43). Line 26 indicates that the method returns a `double` value, that the method's name is `Maximum` and that the method requires *three* `double` parameters (`x`, `y` and `z`) to accomplish its task. When a method has more than one parameter, the parameters are specified as a *comma-separated list*. When `Maximum` is called in line 19, the parameter `x` is initialized with the value of the argument `number1`, the parameter `y` is initialized with the value of the argument `number2` and the parameter `z` is initialized with the value of the argument `number3`. There must be one argument in the method call for each required parameter in the method declaration. Also, each argument must be *consistent* with the type of the corresponding parameter. For example, a parameter of type `double` can receive values like 7.35 (a `double`), 22 (an `int`) or `-0.03456` (a `double`), but not strings like `"hello"`. Section 7.6 discusses the argument types that can be provided in a method call for each parameter of a simple type. Note the use of type `double`'s `Parse` method in lines 12, 14 and 16 to convert into `double` values the strings typed by the user.



#### Common Programming Error 7.2

*Declaring method parameters of the same type as `double x`, `y` instead of `double x`, `double y` is a syntax error—a type is required for each parameter in the parameter list.*

#### *Logic of Determining the Maximum Value*

To determine the maximum value, we begin with the assumption that parameter `x` contains the largest value, so line 28 declares local variable `maximumValue` and initializes it with the value of parameter `x`. Of course, it's possible that parameter `y` or `z` contains the largest value, so we must compare each of these values with `maximumValue`. The `if` statement at lines 31–34 determines whether `y` is greater than `maximumValue`. If so, line 33 assigns `y` to `maximumValue`. The `if` statement at lines 37–40 determines whether `z` is greater than `maximumValue`. If so, line 39 assigns `z` to `maximumValue`. At this point, the largest of the three values resides in `maximumValue`, so line 42 returns that value to line 19 where it's assigned to the variable `result`. When program control returns to the point in the app where `Maximum` was called, `Maximum`'s parameters `x`, `y` and `z` are no longer accessible. Methods can return *at most one* value; the returned value can be a value type that contains one or more values (implemented as a `struct`; Section 10.13) or a reference to an object that contains one or more values.

### 7.4.3 Assembling strings with Concatenation

C# allows string objects to be created by assembling smaller strings into larger strings using operator `+` (or the compound assignment operator `+=`). This is known as **string concatenation**. When both operands of operator `+` are string objects, the `+` operator creates a *new* string object containing copies of the characters in its left operand followed by cop-

ies of the characters in its right operand. For example, the expression "hello " + "there" creates the string "hello there" without disturbing the original strings.

In line 22, the expression "Maximum is: " + result uses operator + with operands of types string and double. Every simple-type value has a string representation. When one of the + operator's operands is a string, the other is implicitly converted to a string, then the two strings are *concatenated*. So, in line 22, the double value is converted to its string representation and placed at the end of "Maximum is: ". If there are any trailing zeros in a double value, these are *discarded*. Thus, the string representation of 9.3500 is "9.35".

### *Anything Can Be Converted to a string*

If a bool is concatenated with a string, the bool is converted to the string "True" or "False" (each is capitalized). In addition, every object has a ToString method that returns a string representation of that object. When an object is concatenated with a string, the object's ToString method is called *implicitly* to obtain the string representation of the object. If the object is null, an *empty string* is written.

If a type does not define a ToString method, the default ToString implementation returns a string containing the type's **fully qualified name**—that is, the namespace in which the type is defined followed by a dot (.) and the type name (e.g., System.Object for the .NET class Object). Each type you create can declare a custom ToString method, as you'll do in Chapter 8 for a Card class that represents a playing card in a deck of cards.

### *Formatting strings with string Interpolation*

Line 22 of Fig. 7.2, of course, could also be written using string interpolation as

```
Console.WriteLine($"Maximum is: {result}");
```

As with string concatenation, using string interpolation to insert an *object* into a string *implicitly* calls the object's ToString method to obtain the object's string representation.

## 7.4.4 Breaking Apart Large string Literals

When a large string literal or interpolated string is typed into an app's source code, you can break that string into several smaller strings and place them on multiple lines for readability. The strings can be reassembled using string concatenation. We discuss the details of strings in Chapter 16.



### Common Programming Error 7.3

*It's a syntax error to break a string literal or interpolated string across multiple lines of code. If a string does not fit on one line, you can split it into several smaller strings and use concatenation to form the desired string.*



### Common Programming Error 7.4

*Confusing the string concatenation + operator with the addition + operator can lead to strange results. The + operator is left-associative. For example, if y has the int value 5, the expression "y + 2 = " + y + 2 results in the string "y + 2 = 52", not "y + 2 = 7", because first the value of y (5) is concatenated with the string "y + 2 = ", then the value 2 is concatenated with the new larger string "y + 2 = 5". The expression "y + 2 = " + (y + 2) produces the desired result "y + 2 = 7". Using C# 6 string interpolation eliminates this problem.*

### 7.4.5 When to Declare Variables as Fields

Variable `result` is a local variable in method `Main` because it's declared in the block that represents the method's body. Variables should be declared as fields of a class (i.e., as either instance variables or `static` variables) *only* if they're required for use in more than one method of the class or if the app should save their values between calls to a given method.

### 7.4.6 Implementing Method `Maximum` by Reusing Method `Math.Max`

Recall from Fig. 7.1 that class `Math`'s `Max` method can determine the larger of two values. The entire body of our maximum method could also be implemented with nested calls to `Math.Max`, as follows:

```
return Math.Max(x, Math.Max(y, z));
```

The leftmost `Math.Max` call has the arguments `x` and `Math.Max(y, z)`. Before any method can be called, the runtime evaluates *all* the arguments to determine their values. If an argument is a method call, the call must be performed to determine its return value. So, in the preceding statement, `Math.Max(y, z)` is evaluated first to determine the larger of `y` and `z`. Then the result is passed as the second argument to the first call to `Math.Max`, which returns the larger of its two arguments. Using `Math.Max` in this manner is a good example of software reuse—we find the largest of three values by reusing `Math.Max`, which finds the larger of two values. Note how concise this code is compared to lines 28–42 of Fig. 7.2.

## 7.5 Notes on Using Methods

### *Three Ways to Call a Method*

You've seen three ways to call a method:

1. Using a method name by itself to call a method of the *same* class—as in line 19 of Fig. 7.2, which calls `Maximum(number1, number2, number3)` from `Main`.
2. Using a reference to an object, followed by the member-access operator (`.`) and the method name to call a non-`static` method of the referenced object—as in line 23 of Fig. 4.12, which called `account1.Deposit(depositAmount)` from the `Main` method of class `AccountTest`.
3. Using the class name and the member-access operator (`.`) to call a `static` method of a class—as in lines 12, 14 and 16 of Fig. 7.2, which each call `Console.ReadLine()`, or as in `Math.Sqrt(900.0)` in Section 7.3.

### *Three Ways to Return from a Method*

You've seen three ways to return control to the statement that calls a method:

- Reaching the method-ending right brace in a method with return type `void`.
- When the following statement executes in a method with return type `void`

```
return;
```

- When a method returns a result with a statement of the following form in which the *expression* is evaluated and its result (and control) are returned to the caller:

```
return expression;
```

**Common Programming Error 7.5**

*Declaring a method outside the body of a class declaration or inside the body of another method is a syntax error.*

**Common Programming Error 7.6**

*Redeclaring a method parameter as a local variable in the method's body is a compilation error.*

**Common Programming Error 7.7**

*Forgetting to return a value from a method that should return one is a compilation error. If a return type other than void is specified, the method must use a return statement to return a value, and that value must be consistent with the method's return type. Returning a value from a method whose return type has been declared void is a compilation error.*

***static Members Can Access Only the Class's Other static Members Directly***

A `static` method or property can call *only* other `static` methods or properties of the same class directly (i.e., using the method name by itself) and can manipulate *only* `static` variables in the same class directly. To access a class's non-`static` members, a `static` method or property must use a reference to an object of that class. Recall that `static` methods relate to a class as a whole, whereas non-`static` methods are associated with a specific object (instance) of the class and may manipulate the instance variables of that object (as well as the class's `static` members).

Many objects of a class, each with its own copies of the instance variables, may exist at the same time. Suppose a `static` method were to invoke a non-`static` method directly. How would the method know which object's instance variables to manipulate? What would happen if no objects of the class existed at the time the non-`static` method was invoked?

**Software Engineering Observation 7.2**

*A static method cannot access non-static members of the same class directly.*

## 7.6 Argument Promotion and Casting

Another important feature of method calls is **argument promotion**—implicitly converting an argument's value to the type that the method expects to receive (if possible) in its corresponding parameter. For example, an app can call `Math.Sqrt` with an integer argument even though the method expects to receive a `double` argument. The statement

```
Console.WriteLine(Math.Sqrt(4));
```

correctly evaluates `Math.Sqrt(4)` and displays the value 2.0. `Sqrt`'s parameter list causes C# to convert the `int` value 4 to the `double` value 4.0 before passing the value to `Sqrt`. Such conversions may lead to compilation errors if C#'s **promotion rules** are not satisfied. The promotion rules specify which conversions are allowed—that is, which conversions can be performed *without losing data*. In the `Sqrt` example above, an `int` is converted to a

`double` without changing its value. However, converting a `double` to an `int` *truncates* the fractional part of the `double` value—thus, part of the value is lost. Also, `double` variables can hold values much larger (and much smaller) than `int` variables, so assigning a `double` to an `int` can cause a loss of information when the `double` value doesn't fit in the `int`. Converting large integer types to small integer types (e.g., `long` to `int`) also can produce incorrect results.

### 7.6.1 Promotion Rules

The promotion rules apply to expressions containing values of two or more simple types and to simple-type values passed as arguments to methods. Each value is promoted to the appropriate type in the expression. (Actually, the expression uses a *temporary* copy of each promoted value—the types of the original values remain unchanged.) Figure 7.3 lists the simple types alphabetically and the types to which each can be promoted. Values of all simple types also can be implicitly converted to type `object`. We demonstrate such implicit conversions in Chapter 19.

Type	Conversion types
<code>bool</code>	no possible implicit conversions to other simple types
<code>byte</code>	<code>ushort</code> , <code>short</code> , <code>uint</code> , <code>int</code> , <code>ulong</code> , <code>long</code> , <code>decimal</code> , <code>float</code> or <code>double</code>
<code>char</code>	<code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> , <code>ulong</code> , <code>decimal</code> , <code>float</code> or <code>double</code>
<code>decimal</code>	no possible implicit conversions to other simple types
<code>double</code>	no possible implicit conversions to other simple types
<code>float</code>	<code>double</code>
<code>int</code>	<code>long</code> , <code>decimal</code> , <code>float</code> or <code>double</code>
<code>long</code>	<code>decimal</code> , <code>float</code> or <code>double</code>
<code>sbyte</code>	<code>short</code> , <code>int</code> , <code>long</code> , <code>decimal</code> , <code>float</code> or <code>double</code>
<code>short</code>	<code>int</code> , <code>long</code> , <code>decimal</code> , <code>float</code> or <code>double</code>
<code>uint</code>	<code>ulong</code> , <code>long</code> , <code>decimal</code> , <code>float</code> or <code>double</code>
<code>ulong</code>	<code>decimal</code> , <code>float</code> or <code>double</code>
<code>ushort</code>	<code>uint</code> , <code>int</code> , <code>ulong</code> , <code>long</code> , <code>decimal</code> , <code>float</code> or <code>double</code>

**Fig. 7.3** | Implicit conversions between simple types.

### 7.6.2 Sometimes Explicit Casts Are Required

By default, C# does not allow you to implicitly convert values between simple types if the target type cannot represent every value of the original type (e.g., the `int` value 2000000 cannot be represented as a `short`, and any floating-point number with nonzero digits after its decimal point cannot be represented in an integer type such as `long`, `int` or `short`).

To prevent a compilation error in cases where information may be lost due to an implicit conversion between simple types, the compiler requires you to use a *cast operator* to *force* the conversion. This enables you to “take control” from the compiler. You essen-

tially say, “I know this conversion might cause loss of information, but for my purposes here, that’s fine.” Suppose you create a method `Square` that calculates the square of an `int` argument. To call `Square` with the whole part of a `double` argument named `doubleValue`, you’d write `Square((int) doubleValue)`. This method call explicitly casts (converts) the value of `doubleValue` to an integer for use in method `Square`. Thus, if `doubleValue`’s value is 4.5, the method receives the value 4 and returns 16, not 20.25.



### Common Programming Error 7.8

*Converting a simple-type value to a value of another simple type may change the value if the promotion is not allowed. For example, converting a floating-point value to an integral value may introduce truncation errors (loss of the fractional part) in the result.*

## 7.7 The .NET Framework Class Library

Many predefined classes are grouped into categories of related classes called *namespaces*. Together, these namespaces are referred to as the .NET Framework Class Library.

### *using Directives and Namespaces*

Throughout the text, `using` directives allow us to use library classes from the Framework Class Library without specifying their namespace names. For example, an app would include the declaration

```
using System;
```

in order to use the class names from the `System` namespace without fully qualifying their names. This allows you to use the *unqualified* name `Console`, rather than the *fully qualified* name `System.Console`, in your code.



### Software Engineering Observation 7.3

*The C# compiler does not require using declarations in a source-code file if the fully qualified class name is specified every time a class name is used. Many programmers prefer the more concise programming style enabled by using declarations.*

You might have noticed in each project containing multiple classes that in each class’s source-code file we did not need additional `using` directives to use the other classes in the project. There’s a special relationship between classes in a project—by default, such classes are in the same namespace and can be used by other classes in the project. Thus, a `using` declaration is not required when one class in a project uses another in the same project—such as when class `AccountTest` used class `Account` in Chapter 4’s examples. Also, any classes that are not *explicitly* placed in a namespace are *implicitly* placed in the so-called **global namespace**.

### *.NET Namespaces*

A strength of C# is the large number of classes in the namespaces of the .NET Framework Class Library. Some key Framework Class Library namespaces are described in Fig. 7.4, which represents only a small portion of the reusable classes in the .NET Framework Class Library.

Namespace	Description
System.Windows.Forms	Contains the classes required to create and manipulate GUIs. (Various classes in this namespace are discussed in Chapter 14, Graphical User Interfaces with Windows Forms: Part 1, and Chapter 15, Graphical User Interfaces with Windows Forms: Part 2.)
System.Windows.Controls System.Windows.Input System.Windows.Media System.Windows.Shapes	Contain the classes of the Windows Presentation Foundation for GUIs, 2-D and 3-D graphics, multimedia and animation.
System.Linq	Contains the classes that support Language Integrated Query (LINQ). (See Chapter 9, Introduction to LINQ and the List Collection, and several other chapters throughout the book.)
System.Data.Entity	Contains the classes for manipulating data in databases (i.e., organized collections of data), including support for LINQ to Entities. (See Chapter 20, Databases and LINQ.)
System.IO	Contains the classes that enable programs to input and output data. (See Chapter 17, Files and Streams.)
System.Web	Contains the classes used for creating and maintaining web apps, which are accessible over the Internet.
System.Xml	Contains the classes for creating and manipulating XML data. Data can be read from or written to XML files.
System.Xml.Linq	Contains the classes that support Language Integrated Query (LINQ) for XML documents. (See Chapter 21, Asynchronous Programming with <code>async</code> and <code>await</code> .)
System.Collections System.Collections.Generic	Contain the classes that define data structures for maintaining collections of data. (See Chapter 19, Generic Collections; Functional Programming with LINQ/PLINQ.)
System.Text	Contains classes that enable programs to manipulate characters and strings. (See Chapter 16, Strings and Characters: A Deeper Look.)

**Fig. 7.4** | .NET Framework Class Library namespaces (a subset).

### *Locating Additional Information About a .NET Class's Methods*

You can locate additional information about a .NET class's methods in the *.NET Framework Class Library* reference

<https://msdn.microsoft.com/library/mt472912>

When you visit this site, you'll see an alphabetical listing of all the namespaces in the Framework Class Library. Locate the namespace and click its link to see an alphabetical listing of all its classes, with a brief description of each. Click a class's link to see a more complete description of the class. Click the **Methods** link in the left-hand column to see a listing of the class's methods.



### Good Programming Practice 7.1

The online .NET Framework documentation is easy to search and provides many details about each class. As you learn each class in this book, you should review it in the online documentation for additional information.

## 7.8 Case Study: Random-Number Generation

In this and the next section, we develop a nicely structured game-playing app with multiple methods. The app uses most of the control statements presented thus far in the book and introduces several new programming concepts.

There's something in the air of a casino that invigorates people—from the high rollers at the plush mahogany-and-felt craps tables to the quarter poppers at the one-armed bandits. It's the **element of chance**, the possibility that luck will convert a pocketful of money into a mountain of wealth. The element of chance can be introduced in an app via an object of class `Random` (of namespace `System`). Objects of class `Random` can produce random `byte`, `int` and `double` values. In the next several examples, we use objects of class `Random` to produce random numbers.

### Secure Random Numbers

According to Microsoft's documentation for class `Random`, the random values it produces “are not completely random because a mathematical algorithm is used to select them, but they are sufficiently random for practical purposes.” Such values should not be used, for example, to create randomly selected passwords. If your app requires so-called cryptographically secure random numbers, use class `RNGCryptoServiceProvider`<sup>1</sup> from namespace `System.Security.Cryptography` to produce random values:

```
https://msdn.microsoft.com/library/system.security.cryptography.rngcryptoserviceprovider
```

### 7.8.1 Creating an Object of Type `Random`

A new random-number generator object can be created with class `Random` (from the `System` namespace) as follows:

```
Random randomNumbers = new Random();
```

The `Random` object can then be used to generate random `byte`, `int` and `double` values—we discuss only random `int` values here.

### 7.8.2 Generating a Random Integer

Consider the following statement:

```
int randomValue = randomNumbers.Next();
```

When called with no arguments, method `Next` of class `Random` generates a random `int` value in the range 0 to +2,147,483,646, inclusive. If the `Next` method truly produces values at random, then every value in that range should have an equal chance (or probability) of being chosen each time method `Next` is called. The values returned by `Next` are actually

1. Class `RNGCryptoServiceProvider` produces arrays of bytes. We discuss arrays in Chapter 8.



**pseudorandom numbers**—a sequence of values produced by a complex mathematical calculation. The calculation uses the current time of day (which, of course, changes constantly) to **seed** the random-number generator such that each execution of an app yields a different sequence of random values.

### 7.8.3 Scaling the Random-Number Range

The range of values produced directly by method `Next` often differs from the range of values required in a particular C# app. For example, an app that simulates coin tossing might require only 0 for “heads” and 1 for “tails.” An app that simulates the rolling of a six-sided die might require random integers in the range 1–6. A video game that randomly predicts the next type of spaceship (out of four possibilities) that will fly across the horizon might require random integers in the range 1–4. For cases like these, class `Random` provides versions of method `Next` that accept arguments. One receives an `int` argument and returns a value from 0 up to, but not including, the argument’s value. For example, you might use the statement

```
int randomValue = randomNumbers.Next(6); // 0, 1, 2, 3, 4 or 5
```

which returns 0, 1, 2, 3, 4 or 5. The argument 6—called the **scaling factor**—represents the number of unique values that `Next` should produce (in this case, six—0, 1, 2, 3, 4 and 5). This manipulation is called **scaling** the range of values produced by `Random` method `Next`.

### 7.8.4 Shifting Random-Number Range

Suppose we wanted to simulate a six-sided die that has the numbers 1–6 on its faces, not 0–5. Scaling the range of values alone is not enough. So we **shift** the range of numbers produced. We could do this by adding a **shifting value**—in this case 1—to the result of method `Next`, as in

```
int face = 1 + randomNumbers.Next(6); // 1, 2, 3, 4, 5 or 6
```

The shifting value (1) specifies the first value in the desired set of random integers. The preceding statement assigns to `face` a random integer in the range 1–6.

### 7.8.5 Combining Shifting and Scaling

The third alternative of method `Next` provides a more intuitive way to express both shifting and scaling. This method receives two `int` arguments and returns a value from the first argument’s value up to, but not including, the second argument’s value. We could use this method to write a statement equivalent to our previous statement, as in

```
int face = randomNumbers.Next(1, 7); // 1, 2, 3, 4, 5 or 6
```

### 7.8.6 Rolling a Six-Sided Die

To demonstrate random numbers, let’s develop an app that simulates 20 rolls of a six-sided die and displays each roll’s value. Figure 7.5 shows two sample outputs, which confirm that the results of the preceding calculation are integers in the range 1–6 and that each run of the app can produce a *different* sequence of random numbers. Line 9 creates the `Random` object `randomNumbers` to produce random values. Line 15 executes 20 times in a loop to roll the die and line 16 displays the value of each roll.

```

1 // Fig. 7.5: RandomIntegers.cs
2 // Shifted and scaled random integers.
3 using System;
4
5 class RandomIntegers
6 {
7     static void Main()
8     {
9         Random randomNumbers = new Random(); // random-number generator
10
11         // loop 20 times
12         for (int counter = 1; counter <= 20; ++counter)
13         {
14             // pick random integer from 1 to 6
15             int face = randomNumbers.Next(1, 7);
16             Console.WriteLine($"{face} "); // display generated value
17         }
18         Console.WriteLine();
19     }
20 }
21 }

```

3 3 3 1 1 2 1 2 4 2 2 3 6 2 5 3 4 6 6 1

6 2 5 1 3 5 2 1 6 5 4 1 6 1 3 3 1 4 3 4

**Fig. 7.5** | Shifted and scaled random integers.

### *Rolling a Six-Sided Die 60,000,000 Times*

To show that the numbers produced by `Next` occur with approximately equal likelihood, let's simulate 60,000,000 rolls of a die (Fig. 7.6). Each integer from 1 to 6 should appear approximately 10,000,000 times.

```

1 // Fig. 7.6: RollDie.cs
2 // Roll a six-sided die 60,000,000 times.
3 using System;
4
5 class RollDie
6 {
7     static void Main()
8     {
9         Random randomNumbers = new Random(); // random-number generator
10
11         int frequency1 = 0; // count of 1s rolled
12         int frequency2 = 0; // count of 2s rolled
13         int frequency3 = 0; // count of 3s rolled
14         int frequency4 = 0; // count of 4s rolled
15         int frequency5 = 0; // count of 5s rolled
16         int frequency6 = 0; // count of 6s rolled

```

**Fig. 7.6** | Roll a six-sided die 60,000,000 times. (Part I of 2.)

```

17
18 // summarize results of 60,000,000 rolls of a die
19 for (int roll = 1; roll <= 60000000; ++roll)
20 {
21     int face = randomNumbers.Next(1, 7); // number from 1 to 6
22
23     // determine roll value 1-6 and increment appropriate counter
24     switch (face)
25     {
26     case 1:
27         ++frequency1; // increment the 1s counter
28         break;
29     case 2:
30         ++frequency2; // increment the 2s counter
31         break;
32     case 3:
33         ++frequency3; // increment the 3s counter
34         break;
35     case 4:
36         ++frequency4; // increment the 4s counter
37         break;
38     case 5:
39         ++frequency5; // increment the 5s counter
40         break;
41     case 6:
42         ++frequency6; // increment the 6s counter
43         break;
44     }
45 }
46
47 Console.WriteLine("Face\tFrequency"); // output headers
48 Console.WriteLine($"1\t{frequency1}\n2\t{frequency2}");
49 Console.WriteLine($"3\t{frequency3}\n4\t{frequency4}");
50 Console.WriteLine($"5\t{frequency5}\n6\t{frequency6}");
51 }
52 }

```

Face	Frequency
1	10006774
2	9993289
3	9993438
4	10006520
5	9998762
6	10001217

Face	Frequency
1	10002183
2	9997815
3	9999619
4	10006012
5	9994806
6	9999565

**Fig. 7.6** | Roll a six-sided die 60,000,000 times. (Part 2 of 2.)

As the two sample outputs show, the values produced by method `Next` enable the app to realistically simulate rolling a six-sided die. The app uses nested control statements (the `switch` is nested inside the `for`) to determine the number of times each side of the die occurred. The `for` statement (lines 19–45) iterates 60,000,000 times. During each iteration, line 21 produces a random value from 1 to 6. This face value is then used as the `switch` expression (line 24). Based on the face value, the `switch` statement increments one of the six counter variables during each iteration of the loop. (In Section 8.4.7, we show an elegant way to replace the entire `switch` statement in this app with a single statement.) The `switch` statement has no default label because we have a case label for every possible die value that the expression in line 21 can produce. Run the app several times and observe the results. You'll see that every time you execute this apk, it produces different results.

### 7.8.7 Scaling and Shifting Random Numbers

Previously, we demonstrated the statement

```
int face = randomNumbers.Next(1, 7);
```

which simulates the rolling of a six-sided die. This statement always assigns to variable `face` an integer in the range  $1 \leq \text{face} < 7$ . The width of this range (i.e., the number of consecutive integers in the range) is 6, and the starting number in the range is 1. Referring to the preceding statement, we see that the width of the range is determined by the difference between the two integers passed to `Random` method `Next`, and the starting number of the range is the value of the first argument. We can generalize this result as

```
int number = randomNumbers.Next(shiftingValue, shiftingValue + scalingFactor);
```

where *shiftingValue* specifies the first number in the desired range of consecutive integers and *scalingFactor* specifies how many numbers are in the range.

It's also possible to choose integers at random from sets of values *other* than ranges of consecutive integers. For this purpose, it's simpler to use the version of the `Next` method that takes only *one* argument. For example, to obtain a random value from the sequence 2, 5, 8, 11 and 14, you could use the statement

```
int number = 2 + 3 * randomNumbers.Next(5);
```

In this case, `randomNumbers.Next(5)` produces values in the range 0–4. Each value produced is multiplied by 3 to produce a number in the sequence 0, 3, 6, 9 and 12. We then add 2 to that value to *shift* the range of values and obtain a value from the sequence 2, 5, 8, 11 and 14. We can generalize this result as

```
int number = shiftingValue +
    differenceBetweenValues * randomNumbers.Next(scalingFactor);
```

where *shiftingValue* specifies the first number in the desired range of values, *differenceBetweenValues* represents the difference between consecutive numbers in the sequence and *scalingFactor* specifies how many numbers are in the range.

### 7.8.8 Repeatability for Testing and Debugging

As we mentioned earlier in this section, the methods of class `Random` actually generate *pseudorandom* numbers based on complex mathematical calculations. Repeatedly calling any of `Random`'s methods produces a sequence of numbers that appears to be random. The cal-

ulation that produces the pseudorandom numbers uses the time of day as a **seed value** to change the sequence’s starting point. Each new `Random` object seeds itself with a value based on the computer system’s clock at the time the object is created, enabling each execution of an app to produce a *different* sequence of random numbers.

When debugging an app, it’s sometimes useful to repeat the *same* sequence of pseudorandom numbers during each execution of the app. This repeatability enables you to prove that your app is working for a specific sequence of random numbers before you test the app with different sequences of random numbers. When repeatability is important, you can create a `Random` object as follows:

```
Random randomNumbers = new Random(seedValue);
```

The `seedValue` argument (an `int`) seeds the random-number calculation—using the *same* `seedValue` every time produces the *same* sequence of random numbers. Different seed values, of course, produce *different* sequences of random numbers.

## 7.9 Case Study: A Game of Chance; Introducing Enumerations

One popular game of chance is the dice game known as “craps,” which is played in casinos and back alleys throughout the world. The rules of the game are straightforward:

*You roll two dice. Each die has six faces, which contain one, two, three, four, five and six spots, respectively. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, you win. If the sum is 2, 3 or 12 on the first throw (called “craps”), you lose (i.e., “the house” wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first throw, that sum becomes your “point.” To win, you must continue rolling the dice until you “make your point” (i.e., roll that same point value). You lose by rolling a 7 before making your point.*

The app in Fig. 7.7 simulates the game of craps, using methods to define the logic of the game. The `Main` method (lines 24–80) calls the static `RollDice` method (lines 83–94) as needed to roll the two dice and compute their sum. The four sample outputs show winning on the first roll, losing on the first roll, losing on a subsequent roll and winning on a subsequent roll, respectively. Variable `randomNumbers` (line 8) is declared `static`, so it can be created once during the program’s execution and used in method `RollDice`.

---

```

1 // Fig. 7.7: Craps.cs
2 // Craps class simulates the dice game craps.
3 using System;
4
5 class Craps
6 {
7     // create random-number generator for use in method RollDice
8     private static Random randomNumbers = new Random();
9
10    // enumeration with constants that represent the game status
11    private enum Status {Continue, Won, Lost}

```

---

**Fig. 7.7** | Craps class simulates the dice game craps. (Part I of 4.)

```
12
13 // enumeration with constants that represent common rolls of the dice
14 private enum DiceNames
15 {
16     SnakeEyes = 2,
17     Trey = 3,
18     Seven = 7,
19     YoLeven = 11,
20     BoxCars = 12
21 }
22
23 // plays one game of craps
24 static void Main()
25 {
26     // gameStatus can contain Continue, Won or Lost
27     Status gameStatus = Status.Continue;
28     int myPoint = 0; // point if no win or loss on first roll
29
30     int sumOfDice = RollDice(); // first roll of the dice
31
32     // determine game status and point based on first roll
33     switch ((DiceNames) sumOfDice)
34     {
35         case DiceNames.Seven: // win with 7 on first roll
36         case DiceNames.YoLeven: // win with 11 on first roll
37             gameStatus = Status.Won;
38             break;
39         case DiceNames.SnakeEyes: // lose with 2 on first roll
40         case DiceNames.Trey: // lose with 3 on first roll
41         case DiceNames.BoxCars: // lose with 12 on first roll
42             gameStatus = Status.Lost;
43             break;
44         default: // did not win or lose, so remember point
45             gameStatus = Status.Continue; // game is not over
46             myPoint = sumOfDice; // remember the point
47             Console.WriteLine($"Point is {myPoint}");
48             break;
49     }
50
51     // while game is not complete
52     while (gameStatus == Status.Continue) // game not Won or Lost
53     {
54         sumOfDice = RollDice(); // roll dice again
55
56         // determine game status
57         if (sumOfDice == myPoint) // win by making point
58         {
59             gameStatus = Status.Won;
60         }
61         else
62         {
```

**Fig. 7.7** | Craps class simulates the dice game craps. (Part 2 of 4.)

```
63         // lose by rolling 7 before point
64         if (sumOfDice == (int) DiceNames.Seven)
65         {
66             gameState = Status.Lost;
67         }
68     }
69 }
70
71 // display won or lost message
72 if (gameStatus == Status.Won)
73 {
74     Console.WriteLine("Player wins");
75 }
76 else
77 {
78     Console.WriteLine("Player loses");
79 }
80 }
81
82 // roll dice, calculate sum and display results
83 static int RollDice()
84 {
85     // pick random die values
86     int die1 = randomNumbers.Next(1, 7); // first die roll
87     int die2 = randomNumbers.Next(1, 7); // second die roll
88
89     int sum = die1 + die2; // sum of die values
90
91     // display results of this roll
92     Console.WriteLine($"Player rolled {die1} + {die2} = {sum}");
93     return sum; // return sum of dice
94 }
95 }
```

```
Player rolled 2 + 5 = 7
Player wins
```

```
Player rolled 2 + 1 = 3
Player loses
```

```
Player rolled 2 + 4 = 6
Point is 6
Player rolled 3 + 1 = 4
Player rolled 5 + 5 = 10
Player rolled 6 + 1 = 7
Player loses
```

**Fig. 7.7** | Craps class simulates the dice game craps. (Part 3 of 4.)

```

Player rolled 4 + 6 = 10
Point is 10
Player rolled 1 + 3 = 4
Player rolled 1 + 3 = 4
Player rolled 2 + 3 = 5
Player rolled 4 + 4 = 8
Player rolled 6 + 6 = 12
Player rolled 4 + 4 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 6 = 8
Player rolled 6 + 6 = 12
Player rolled 6 + 4 = 10
Player wins

```

**Fig. 7.7** | Craps class simulates the dice game craps. (Part 4 of 4.)

### 7.9.1 Method RollDice

In the rules of the game, the player must roll two dice on the first roll and must do the same on all subsequent rolls. We declare method `RollDice` (lines 83–94) to roll the dice and compute and display their sum. Method `RollDice` is declared once, but it’s called from two places (lines 30 and 54) in method `Main`, which contains the logic for one complete game of craps. Method `RollDice` takes no arguments, so it has an empty parameter list. Each time it’s called, `RollDice` returns the sum of the dice as an `int`. Although lines 86 and 87 look the same (except for the die names), they do not necessarily produce the same result. Each of these statements produces a random value in the range 1–6. Variable `randomNumbers` (used in lines 86–87) is *not* declared in the method. Rather it’s declared as a `private static` variable of the class and initialized in line 8. This enables us to create *one* `Random` object that’s reused in each call to `RollDice`.

### 7.9.2 Method Main’s Local Variables

The game is reasonably involved. The player may win or lose on the first roll or may win or lose on any subsequent roll. Method `Main` (lines 24–80) uses local variable `gameStatus` (line 27) to keep track of the overall game status, local variable `myPoint` (line 28) to store the “point” if the player does not win or lose on the first roll and local variable `sumOfDice` (line 30) to maintain the sum of the dice for the most recent roll. Variable `myPoint` is initialized to 0 to ensure that the app will compile. If you do not initialize `myPoint`, the compiler issues an error, because `myPoint` is not assigned a value in every case of the `switch` statement—thus, the app could try to use `myPoint` before it’s definitely assigned a value. By contrast, `gameStatus` does not require initialization because it’s assigned a value in every branch of the `switch` statement—thus, it’s guaranteed to be initialized before it’s used. However, as good practice, we initialize it anyway.

### 7.9.3 enum Type Status

Local variable `gameStatus` (line 27) is declared to be of a new type called `Status`, which we declared in line 11. `Status` is a user-defined type called an **enumeration**, which declares a set of *constants* represented by identifiers. An enumeration is introduced by the keyword **enum** and a type name (in this case, `Status`). As with a class, braces (`{` and `}`) delimit the



body of an enum declaration. Inside the braces is a comma-separated list of **enumeration constants**—by default, the first constant has the value 0 and each subsequent constant’s value is incremented by 1. The enum constant names must be *unique*, but the value associated with each constant need not be. Type `Status` is declared as a private member of class `Craps`, because `Status` is used only in that class.

Variables of type `Status` should be assigned only one of the three constants declared in the enumeration. When the game is won, the app sets local variable `gameStatus` to `Status.Won` (lines 37 and 59). When the game is lost, the app sets `gameStatus` to `Status.Lost` (lines 42 and 66). Otherwise, the app sets `gameStatus` to `Status.Continue` (line 45) to indicate that the dice must be rolled again.



### Good Programming Practice 7.2

*Using enumeration constants (like `Status.Won`, `Status.Lost` and `Status.Continue`) rather than literal integer values (such as 0, 1 and 2) can make code easier to read and maintain.*

## 7.9.4 The First Roll

Line 30 in method `Main` calls `RollDice`, which picks two random values from 1 to 6, displays the value of the first die, the value of the second die and the sum of the dice, and returns the sum of the dice. Method `Main` next enters the `switch` statement at lines 33–49, which uses the `sumOfDice` value to determine whether the game has been won or lost, or whether it should continue with another roll.

## 7.9.5 enum Type DiceNames

The sums of the dice that would result in a win or loss on the first roll are declared in the `DiceNames` enumeration in lines 14–21. These are used in the `switch` statement’s cases. The identifier names use casino parlance for these sums. In the `DiceNames` enumeration, we assign a value explicitly to each identifier name. When the enum is declared, each constant in the enum declaration is a constant value of type `int`. If you do not assign a value to an identifier in the enum declaration, the compiler will do so. If the first enum constant is unassigned, the compiler gives it the value 0. If any other enum constant is unassigned, the compiler gives it a value one higher than that of the preceding enum constant. For example, in the `Status` enumeration, the compiler implicitly assigns 0 to `Status.Continue`, 1 to `Status.Won` and 2 to `Status.Lost`.

## 7.9.6 Underlying Type of an enum

You could also declare an enum’s underlying type to be `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` or `ulong` by writing

```
private enum MyEnum : typeName {Constant1, Constant2, ...}
```

where `typeName` represents one of the integral simple types.

## 7.9.7 Comparing Integers and enum Constants

If you need to compare a simple integral type value to the underlying value of an enumeration constant, you must use a cast operator to make the two types match—there are no implicit conversions between enum and integral types. In the `switch` expression (line 33),

we use the cast operator to convert the `int` value in `sumOfDice` to type `DiceNames` and compare it to each of the constants in `DiceNames`. Lines 35–36 determine whether the player won on the first roll with `Seven` (7) or `YoLeven` (11). Lines 39–41 determine whether the player lost on the first roll with `SnakeEyes` (2), `Trey` (3) or `BoxCars` (12). After the first roll, if the game is not over, the `default` case (lines 44–48) saves `sumOfDice` in `myPoint` (line 46) and displays the point (line 47).

### *Additional Rolls of the Dice*

If we're still trying to “make our point” (i.e., the game is continuing from a prior roll), the loop in lines 52–69 executes. Line 54 rolls the dice again. If `sumOfDice` matches `myPoint` in line 57, line 59 sets `gameStatus` to `Status.Won`, and the loop terminates because the game is complete. In line 64, we use the cast operator (`int`) to obtain the underlying value of `DiceNames.Seven` so that we can compare it to `sumOfDice`. If `sumOfDice` is equal to `Seven` (7), line 66 sets `gameStatus` to `Status.Lost`, and the loop terminates because the game is over. When the game completes, lines 72–79 display a message indicating whether the player won or lost, and the app terminates.

### *Control Statements in the Craps Example*

Note the use of the various program-control mechanisms we've discussed. The `Craps` class uses two methods—`Main` and `RollDice` (called twice from `Main`)—and the `switch`, `while`, `if...else` and nested `if` control statements. Also, notice that we use multiple case labels in the `switch` statement to execute the same statements for sums of `Seven` and `YoLeven` (lines 35–36) and for sums of `SnakeEyes`, `Trey` and `BoxCars` (lines 39–41).

### *Code Snippets for Auto-Implemented Properties*

Visual Studio has a feature called **code snippets** that allows you to insert *predefined code templates* into your source code. One such snippet enables you to easily create a `switch` statement with cases for all possible values for an `enum` type. Type `switch` in the C# code then press *Tab* twice. If you specify a variable of an `enum` type in the `switch` statement's expression and press *Enter*, a case for each `enum` constant will be generated automatically.

To get a list of all available code snippets, type *Ctrl + k, Ctrl + x*. This displays the **Insert Snippet** window in the code editor. You can navigate through the Visual C# snippet folders with the mouse to see the snippets. This feature also can be accessed by *right clicking* in the source code editor and selecting the **Insert Snippet...** menu item.

## 7.10 Scope of Declarations

You've seen declarations of C# entities, such as classes, methods, properties, variables and parameters. Declarations introduce names that can be used to refer to such C# entities. The **scope** of a declaration is the portion of the app that can refer to the declared entity by its unqualified name. Such an entity is said to be “in scope” for that portion of the app. This section introduces several important scope issues. The basic scope rules are as follows:

1. The scope of a parameter declaration is the body of the method in which the declaration appears.
2. The scope of a local-variable declaration is from the point at which the declaration appears to the end of the block containing the declaration.

3. The scope of a local-variable declaration that appears in the initialization section of a `for` statement's header is the body of the `for` statement and the other expressions in the header.
4. The scope of a method, property or field of a class is the entire body of the class. This enables non-static methods and properties of a class to use any of the class's fields, methods and properties, regardless of the order in which they're declared. Similarly, static methods and properties can use any of the static members of the class.

Any block may contain variable declarations. If a local variable or parameter in a method has the same name as a field, the field is hidden until the block terminates—in Chapter 10, we discuss how to access hidden fields. A compilation error occurs if a *nested block* in a method contains a variable with the same name as a local variable in an *outer block* of the method. The app in Fig. 7.8 demonstrates scoping issues with fields and local variables.



### Error-Prevention Tip 7.1

*Use different names for fields and local variables to help prevent subtle logic errors that occur when a method is called and a local variable of the method hides a field of the same name in the class.*

```

1 // Fig. 7.8: Scope.cs
2 // Scope class demonstrates static- and local-variable scopes.
3 using System;
4
5 class Scope
6 {
7     // static variable that's accessible to all methods of this class
8     private static int x = 1;
9
10    // Main creates and initializes local variable x
11    // and calls methods UseLocalVariable and UseStaticVariable
12    static void Main()
13    {
14        int x = 5; // method's local variable x hides static variable x
15
16        Console.WriteLine($"local x in method Main is {x}");
17
18        // UseLocalVariable has its own local x
19        UseLocalVariable();
20
21        // UseStaticVariable uses class Scope's static variable x
22        UseStaticVariable();
23
24        // UseLocalVariable reinitializes its own local x
25        UseLocalVariable();
26
27        // class Scope's static variable x retains its value
28        UseStaticVariable();

```

**Fig. 7.8** | Scope class demonstrates static- and local-variable scopes. (Part 1 of 2.)

```

29
30     Console.WriteLine($"\\nlocal x in method Main is {x}");
31 }
32
33 // create and initialize local variable x during each call
34 static void UseLocalVariable()
35 {
36     int x = 25; // initialized each time UseLocalVariable is called
37
38     Console.WriteLine(
39         $"\\nlocal x on entering method UseLocalVariable is {x}");
40     ++x; // modifies this method's local variable x
41     Console.WriteLine(
42         $"local x before exiting method UseLocalVariable is {x}");
43 }
44
45 // modify class Scope's static variable x during each call
46 static void UseStaticVariable()
47 {
48     Console.WriteLine("\\nstatic variable x on entering method " +
49         $"UseStaticVariable is {x}");
50     x *= 10; // modifies class Scope's static variable x
51     Console.WriteLine("static variable x before exiting " +
52         $"method UseStaticVariable is {x}");
53 }
54 }

```

```

local x in method Main is 5

local x on entering method UseLocalVariable is 25
local x before exiting method UseLocalVariable is 26

static variable x on entering method UseStaticVariable is 1
static variable x before exiting method UseStaticVariable is 10

local x on entering method UseLocalVariable is 25
local x before exiting method UseLocalVariable is 26

static variable x on entering method UseStaticVariable is 10
static variable x before exiting method UseStaticVariable is 100

local x in method Main is 5

```

**Fig. 7.8** | Scope class demonstrates static- and local-variable scopes. (Part 2 of 2.)

Line 8 declares and initializes the static variable `x` to 1. This static variable is *hidden* in any block (or method) that declares a local variable named `x`. Method `Main` (lines 12–31) declares local variable `x` (line 14) and initializes it to 5. This local variable’s value is output to show that static variable `x` (whose value is 1) is hidden in method `Main`. The app declares two other methods—`UseLocalVariable` (lines 34–43) and `UseStaticVariable` (lines 46–53)—that each take no arguments and do not return results. Method `Main` calls each method twice (lines 19–28). Method `UseLocalVariable` declares local variable `x` (line 36). When `UseLocalVariable` is first called (line 19), it creates local variable `x` and

initializes it to 25 (line 36), outputs the value of `x` (lines 38–39), increments `x` (line 40) and outputs the value of `x` again (lines 41–42). When `UseLocalVariable` is called a second time (line 25), it re-creates local variable `x` and reinitializes it to 25, so the output of each call to `UseLocalVariable` is identical.

Method `UseStaticVariable` does not declare any local variables. Therefore, when it refers to `x`, static variable `x` (line 8) of the class is used. When method `UseStaticVariable` is first called (line 22), it outputs the value (1) of static variable `x` (lines 48–49), multiplies the static variable `x` by 10 (line 50) and outputs the value (10) of static variable `x` again (lines 51–52) before returning. The next time method `UseStaticVariable` is called (line 28), the static variable has its modified value, 10, so the method outputs 10, then 100. Finally, in method `Main`, the app outputs the value of local variable `x` again (line 30) to show that none of the method calls modified `Main`'s local variable `x`, because the methods all referred to variables named `x` in other scopes.

## 7.11 Method-Call Stack and Activation Records

To understand how C# performs method calls, we first need to consider a data structure (i.e., collection of related data items) known as a **stack**. Think of a stack as analogous to a pile of dishes. When a dish is placed on the pile, it's placed at the *top*—referred to as **pushing** the dish onto the stack. Similarly, when a dish is removed from the pile, it's removed from the top—referred to as **popping** the dish off the stack. Stacks are known as **last-in, first-out (LIFO) data structures**—the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.

### 7.11.1 Method-Call Stack

The **method-call stack** (sometimes referred to as the **program-execution stack**) is a data structure that works behind the scenes to support the method call/return mechanism. It also supports the creation, maintenance and destruction of each called method's local variables. As we'll see in Figs. 7.10–7.12, the stack's last-in, first-out (LIFO) behavior is *exactly* what a method needs in order to return to the method that called it.

### 7.11.2 Stack Frames

As each method is called, it may, in turn, call other methods, which may, in turn, call other methods—all *before* any of the methods return. Each method eventually must return control to the method that called it. So, somehow, the system must keep track of the *return addresses* that each method needs in order to return control to the method that called it. The method-call stack is the perfect data structure for handling this information. Each time a method calls another method, an entry is *pushed* onto the stack. This entry, called a **stack frame** or an **activation record**, contains the *return address* that the called method needs in order to return to the calling method. It also contains some additional information we'll soon discuss. If the called method returns instead of calling another method before returning, the stack frame for the method call is *popped*, and control transfers to the return address in the popped stack frame. The same techniques apply when a method accesses a property or when a property calls a method.

The beauty of the call stack is that each called method *always* finds the information it needs to return to its caller at the *top* of the call stack. And, if a method makes a call to

another method, a stack frame for the new method call is simply *pushed* onto the call stack. Thus, the return address required by the newly called method to return to its caller is now located at the *top* of the stack.

### 7.1.1.3 Local Variables and Stack Frames

The stack frames have another important responsibility. Most methods have local variables—parameters and any local variables the method declares. Local variables need to exist while a method is executing. They need to remain active if the method makes calls to other methods. But when a called method returns to its caller, the called method’s local variables need to “go away.” The called method’s stack frame is a perfect place to reserve the memory for the called method’s local variables. That stack frame exists as long as the called method is active. When that method returns—and no longer needs its local variables—its stack frame is *popped* from the stack, and those local variables no longer exist.

### 7.1.1.4 Stack Overflow

Of course, the amount of memory in a computer is finite, so only a certain amount of memory can be used to store activation records on the method-call stack. If more method calls occur than can have their activation records stored on the method-call stack, a fatal error known as **stack overflow** occurs<sup>2</sup>—typically caused by infinite recursion (Section 7.16).

### 7.1.1.5 Method-Call Stack in Action

Now let’s consider how the call stack supports the operation of a `Square` method (lines 15–18 of Fig. 7.9) called by `Main` (lines 8–12).

---

```

1 // Fig. 7.9: SquareTest.cs
2 // Square method used to demonstrate the method
3 // call stack and activation records.
4 using System;
5
6 class Program
7 {
8     static void Main()
9     {
10         int x = 10; // value to square (local variable in main)
11         Console.WriteLine($"x squared: {Square(x)}");
12     }
13
14     // returns the square of an integer
15     static int Square(int y) // y is a local variable
16     {
17         return y * y; // calculate square of y and return result
18     }
19 }
```

---

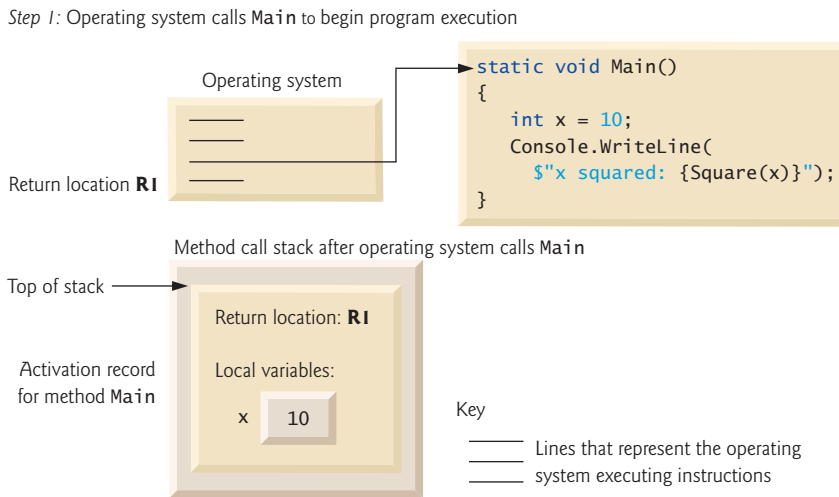
**Fig. 7.9** | Square method used to demonstrate the method-call stack and activation records. (Part 1 of 2.)

2. This is how the website [stackoverflow.com](http://stackoverflow.com) got its name. This is a popular website for getting answers to your programming questions.

```
x squared: 100
```

**Fig. 7.9** | Square method used to demonstrate the method-call stack and activation records. (Part 2 of 2.)

First, the operating system calls `Main`—this *pushes* an activation record onto the stack (Fig. 7.10). This tells `Main` how to return to the operating system (i.e., transfer to return address `R1`) and contains the space for `Main`'s local variable `x`, which is initialized to 10.



**Fig. 7.10** | Method-call stack after the operating system calls `main` to execute the program.

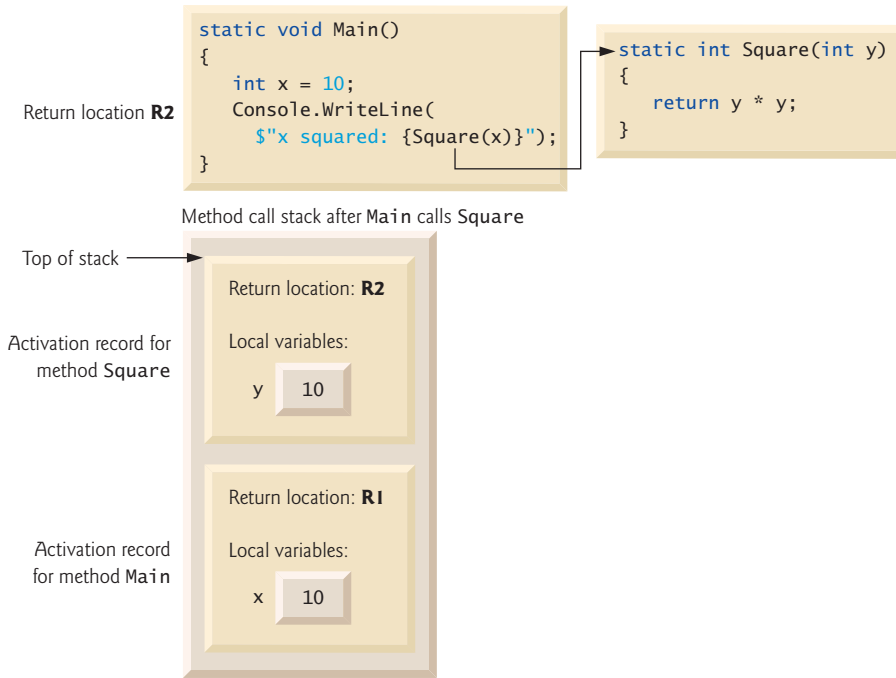
Method `Main`—before returning to the operating system—calls method `Square` in line 11 of Fig. 7.9. This causes a stack frame for `Square` (lines 15–18) to be pushed onto the method-call stack (Fig. 7.11). This stack frame contains the return address that `Square` needs to return to `Main` (i.e., `R2`) and the memory for `Square`'s local variable `y`.

After `Square` performs its calculation, it needs to return to `Main`—and no longer needs the memory for `y`. So `Square`'s stack frame is *popped* from the stack—giving `Square` the return location in `Main` (i.e., `R2`) and losing `Square`'s local variable (Step 3). Figure 7.12 shows the method-call stack *after* `Square`'s activation record has been *popped*.

Method `Main` now displays the result of calling `Square` (Fig. 7.9, line 11). Reaching the closing right brace of `Main` causes its stack frame to be *popped* from the stack, giving `Main` the address it needs to return to the operating system (i.e., `R1` in Fig. 7.10)—at this point, `Main`'s local variable `x` no longer exists.

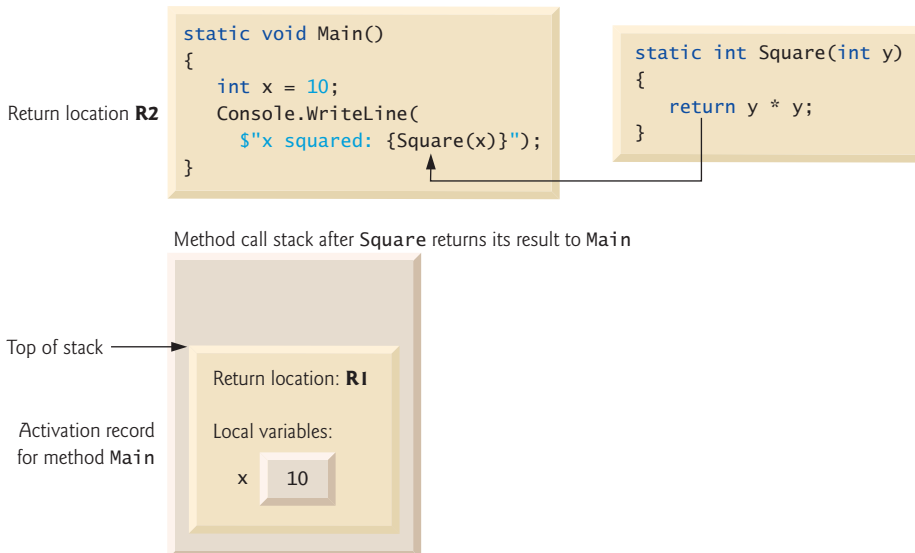
You've now seen how valuable the stack data structure is in implementing a key mechanism that supports program execution. There's a significant omission in the sequence of illustrations in this section. See if you can spot it before reading the next sentence. The call to the method `Console.WriteLine`, of course, also involves the stack, which should be reflected in this section's illustrations and discussion.

Step 2: Main calls method Square to perform calculation



**Fig. 7.11** | Method-call stack after `Main` calls `square` to perform the calculation.

Step 3: `Square` returns its result to `Main`



**Fig. 7.12** | Method-call stack after method `square` returns to `Main`.



## 7.12 Method Overloading

Methods of the same name can be declared in the same class, as long as they have different sets of parameters (determined by the number, types and order of the parameters). This is called **method overloading**. When an **overloaded method** is called, the C# compiler selects the appropriate method by examining the number, types and order of the arguments in the call. Method overloading is commonly used to create several methods with the *same name* that perform the same or similar tasks, but on *different types* or *different numbers of arguments*. For example, Random method Next (Section 7.8) has overloads that accept different numbers of arguments, and Math method Max has overloads that accept different types of arguments (ints vs. doubles). These find the minimum and maximum, respectively, of two values of each of the numeric simple types. Our next example demonstrates declaring and invoking overloaded methods. You'll see examples of overloaded constructors in Chapter 10.

### 7.12.1 Declaring Overloaded Methods

In class MethodOverload (Fig. 7.13), we include two Square methods—one that calculates the square of an int (and returns an int) and one that calculates the square of a double (and returns a double). Although these methods have the same name and similar parameter lists and bodies, you can think of them simply as *different* methods. It may help to think of the method names as “Square of int” and “Square of double,” respectively.

---

```

1 // Fig. 7.13: MethodOverload.cs
2 // Overloaded method declarations.
3 using System;
4
5 class MethodOverload
6 {
7     // test overloaded square methods
8     static void Main()
9     {
10        Console.WriteLine($"Square of integer 7 is {Square(7)}");
11        Console.WriteLine($"Square of double 7.5 is {Square(7.5)}");
12    }
13
14    // square method with int argument
15    static int Square(int intValue)
16    {
17        Console.WriteLine($"Called square with int argument: {intValue}");
18        return intValue * intValue;
19    }
20
21    // square method with double argument
22    static double Square(double doubleValue)
23    {
24        Console.WriteLine(
25            $"Called square with double argument: {doubleValue}");
26        return doubleValue * doubleValue;
27    }
28 }

```

---

**Fig. 7.13** | Overloaded method declarations. (Part 1 of 2.)

```
Called square with int argument: 7
Square of integer 7 is 49
Called square with double argument: 7.5
Square of double 7.5 is 56.25
```

**Fig. 7.13** | Overloaded method declarations. (Part 2 of 2.)

Line 10 in `Main` invokes method `Square` with the argument `7`. Literal integer values are treated as type `int`, so the method call in line 10 invokes the version of `Square` at lines 15–19 that specifies an `int` parameter. Similarly, line 11 invokes method `Square` with the argument `7.5`. Literal real-number values are treated as type `double`, so the method call in line 11 invokes the version of `Square` at lines 22–27 that specifies a `double` parameter. Each method first outputs a line of text to prove that the proper method was called in each case.

The overloaded methods in Fig. 7.13 perform the same calculation, but with two different types. C#'s generics feature provides a mechanism for writing a single “generic method” that can perform the same tasks as an entire set of overloaded methods. We discuss generic methods in Chapter 18.

### 7.12.2 Distinguishing Between Overloaded Methods

The compiler distinguishes overloaded methods by their **signature**—a combination of the method's name and the number, types and order of its parameters. The signature also includes the way those parameters are passed, which can be modified by the `ref` and `out` keywords (discussed in Section 7.18). If the compiler looked only at method names during compilation, the code in Fig. 7.13 would be *ambiguous*—the compiler would not know how to distinguish between the `Square` methods (lines 15–19 and 22–27). Internally, the compiler uses signatures to determine whether a class's methods are unique in that class.

For example, in Fig. 7.13, the compiler will use the method signatures to distinguish between the “Square of `int`” method (the `Square` method that specifies an `int` parameter) and the “Square of `double`” method (the `Square` method that specifies a `double` parameter). As another example, if `Method1`'s declaration begins as

```
void Method1(int a, float b)
```

then that method will have a different signature than a method that begins with

```
void Method1(float a, int b)
```

The *order* of the parameter types is important—the compiler considers the preceding two `Method1` headers to be distinct.

### 7.12.3 Return Types of Overloaded Methods

In discussing the logical names of methods used by the compiler, we did not mention the methods' return types. Methods *cannot* be distinguished by return type. If in a class named `MethodOverloadError` you define overloaded methods with the following headers:

```
int Square(int x)
double Square(int x)
```

which each have the *same* signature but *different* return types, the compiler generates the following error for the second Square method:

```
Type 'MethodOverloadError' already defines a member called 'Square'
with the same parameter types
```

Overloaded methods can have the *same* or *different* return types if the parameter lists are *different*. Also, overloaded methods need not have the same number of parameters.



### Common Programming Error 7.9

*Declaring overloaded methods with identical parameter lists is a compilation error regardless of whether the return types are different.*

## 7.13 Optional Parameters

Methods can have **optional parameters** that allow the calling method to *vary the number of arguments* to pass. An optional parameter specifies a **default value** that's assigned to the parameter if the optional argument is omitted. You can create methods with one or more optional parameters. *All optional parameters must be placed to the right of the method's non-optional parameters*—that is, at the end of the parameter list.



### Common Programming Error 7.10

*Declaring a non-optional parameter to the right of an optional one is a compilation error.*

When a parameter has a default value, the caller has the *option* of passing that particular argument. For example, the method header

```
static int Power(int baseValue, int exponentValue = 2)
```

specifies an optional second parameter. Each call to `Power` must pass at least a `baseValue` argument, or a compilation error occurs. Optionally, a second argument (for the `exponentValue` parameter) can be passed to `Power`. Each optional parameter must specify a default value by using an equal (=) sign followed by the value. For example, the header for `Power` sets 2 as `exponentValue`'s default value. Consider the following calls to `Power`:

- `Power()`—This call generates a compilation error because this method requires a minimum of one argument.
- `Power(10)`—This call is valid because one argument (10) is being passed. The optional `exponentValue` is not specified in the method call, so the compiler uses 2 for the `exponentValue`, as specified in the method header.
- `Power(10, 3)`—This call is also valid because 10 is passed as the required argument and 3 is passed as the optional argument.

Figure 7.14 demonstrates an optional parameter. The program calculates the result of raising a base value to an exponent. Method `Power` (lines 15–25) specifies that its second parameter is optional. In `Main`, lines 10–11 call method `Power`. Line 10 calls the method without the optional second argument. In this case, the compiler provides the second argument, 2, using the default value of the optional argument, which is not visible to you in the call.

```

1 // Fig. 7.14: CalculatePowers.cs
2 // Optional parameter demonstration with method Power.
3 using System;
4
5 class CalculatePowers
6 {
7     // call Power with and without optional arguments
8     static void Main()
9     {
10        Console.WriteLine($"Power(10) = {Power(10)}");
11        Console.WriteLine($"Power(2, 10) = {Power(2, 10)}");
12    }
13
14    // use iteration to calculate power
15    static int Power(int baseValue, int exponentValue = 2)
16    {
17        int result = 1;
18
19        for (int i = 1; i <= exponentValue; ++i)
20        {
21            result *= baseValue;
22        }
23
24        return result;
25    }
26 }

```

```

Power(10) = 100
Power(2, 10) = 1024

```

**Fig. 7.14** | Optional parameter demonstration with method `Power`.

## 7.14 Named Parameters

Normally, when calling a method, the argument values—in order—are assigned to the parameters from left to right in the parameter list. Consider a `Time` class that stores the time of day in 24-hour clock format as `int` values representing the hour (0–23), minute (0–59) and second (0–59). Such a class might provide a `SetTime` method with optional parameters like

```
public void SetTime(int hour = 0, int minute = 0, int second = 0)
```

In the preceding method header, all three of `SetTime`'s parameters are optional. Assuming that we have a `Time` object named `t`, consider the following calls to `SetTime`:

- `t.SetTime()`—This call specifies no arguments, so the compiler assigns the default value 0 to each parameter. The resulting time is 12:00:00 AM.
- `t.SetTime(12)`—This call specifies the argument 12 for the first parameter, hour, and the compiler assigns the default value 0 to the minute and second parameters. The resulting time is 12:00:00 PM.
- `t.SetTime(12, 30)`—This call specifies the arguments 12 and 30 for the parameters hour and minute, respectively, and the compiler assigns the default value 0 to the parameter second. The resulting time is 12:30:00 PM.

- `t.SetTime(12, 30, 22)`—This call specifies the arguments 12, 30 and 22 for the parameters `hour`, `minute` and `second`, respectively, so the compiler does not provide any default values. The resulting time is 12:30:22 PM.

What if you wanted to specify only arguments for the hour and second? You might think that you could call the method as follows:

```
t.SetTime(12, , 22); // COMPILATION ERROR
```

C# doesn't allow you to skip an argument as shown above. C# provides a feature called **named parameters**, which enable you to call methods that receive optional parameters by providing *only* the optional arguments you wish to specify. To do so, you explicitly specify the parameter's name and value—separated by a colon (`:`)—in the argument list of the method call. For example, the preceding statement can be written as follows:

```
t.SetTime(hour: 12, second: 22); // sets the time to 12:00:22
```

In this case, the compiler assigns parameter `hour` the argument 12 and parameter `second` the argument 22. The parameter `minute` is not specified, so the compiler assigns it the default value 0. It's also possible to specify the arguments *out of order* when using named parameters. The arguments for the required parameters must always be supplied. The *argumentName: value* syntax may be used with any method's required parameters.

## 6 7.15 C# 6 Expression-Bodied Methods and Properties

C# 6 introduces a new concise syntax for:

- methods that contain only a return statement that returns a value
- read-only properties in which the `get` accessor contains only a return statement
- methods that contain single statement bodies.

Consider the following `Cube` method:

```
static int Cube(int x)
{
    return x * x * x;
}
```

In C# 6, this can be expressed with an **expression-bodied method** as

```
static int Cube(int x) => x * x * x;
```

The value of `x * x * x` is returned to `Cube`'s caller implicitly. The symbol `=>` follows the method's parameter list and introduces the method's body—no braces or return statement are required and this can be used with `static` and non-`static` methods alike. If the expression to the right of `=>` does not have a value (e.g., a call to a method that returns `void`), the expression-bodied method must return `void`. Similarly, a read-only property can be implemented as an **expression-bodied property**. The following reimplements the `IsNoFaultState` property in Fig. 6.11 to return the result of a logical expression:

```
public bool IsNoFaultState =>
    State == "MA" || State == "NJ" || State == "NY" || State == "PA";
```

## 7.16 Recursion

The apps we've discussed thus far are generally structured as methods that call one another in a disciplined, hierarchical manner. For some problems, however, it's useful to have a method call itself. A **recursive method** is a method that calls itself, either *directly* or *indirectly through another method*. We consider recursion conceptually first. Then we examine an app containing a recursive method.

### 7.16.1 Base Cases and Recursive Calls

Recursive problem-solving approaches have a number of elements in common. When a recursive method is called to solve a problem, it actually is capable of solving *only* the simplest case(s), or **base case(s)**. If the method is called with a base case, it returns a result. If the method is called with a more complex problem, it divides the problem into two conceptual pieces (often called *divide and conquer*): a piece that the method knows how to do and a piece that it does not know how to do. To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or slightly smaller version of it. Because this new problem looks like the original problem, the method calls a fresh copy (or several fresh copies) of itself to work on the smaller problem; this is referred to as a **recursive call** and is also called the **recursion step**. The recursion step normally includes a return statement, because its result will be combined with the portion of the problem the method knew how to solve to form a result that will be passed back to the original caller.

The recursion step executes while the original call to the method is still active (i.e., while it has not finished executing). The recursion step can result in many more recursive calls, as the method divides each new subproblem into two conceptual pieces. For the recursion to *terminate* eventually, each time the method calls itself with a slightly simpler version of the original problem, the sequence of smaller and smaller problems must converge on the base case(s). At that point, the method recognizes the base case and returns a result to the previous copy of the method. A sequence of returns ensues until the original method call returns the result to the caller. This process sounds complex compared with the conventional problem solving we've performed to this point.

### 7.16.2 Recursive Factorial Calculations

Let's write a recursive app to perform a popular mathematical calculation. The factorial of a nonnegative integer  $n$ , written  $n!$  (and pronounced " $n$  factorial"), is the product

$$n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1$$

$1!$  is equal to 1 and  $0!$  is defined to be 1. For example,  $5!$  is the product  $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ , which is equal to 120.

The factorial of an integer, number, greater than or equal to 0 can be calculated iteratively (nonrecursively) using the `for` statement as follows:

```
long factorial = 1;

for (long counter = number; counter >= 1; --counter)
{
    factorial *= counter;
}
```

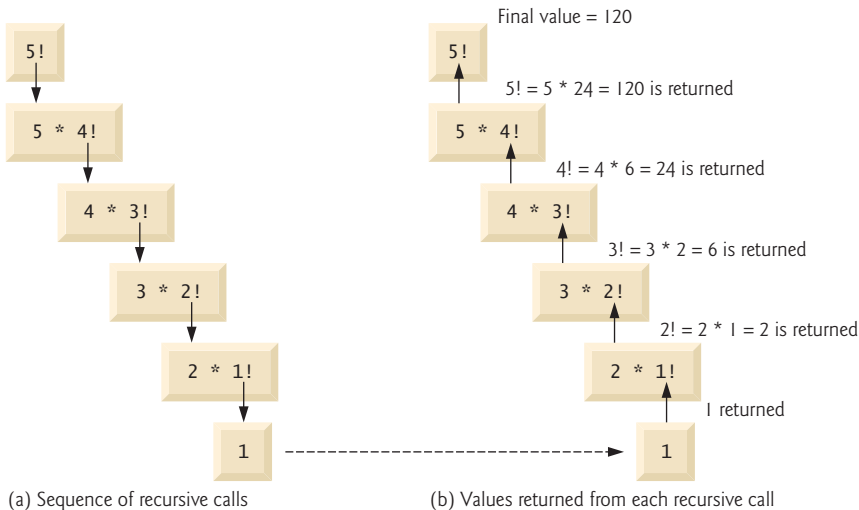
A recursive declaration of the factorial method is arrived at by observing the following relationship:

$$n! = n \cdot (n - 1)!$$

For example,  $5!$  is clearly equal to  $5 \cdot 4!$ , as is shown by the following equations:

$$\begin{aligned} 5! &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \\ 5! &= 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) \\ 5! &= 5 \cdot (4!) \end{aligned}$$

The evaluation of  $5!$  would proceed as shown in Fig. 7.15. Figure 7.15(a) shows how the succession of recursive calls proceeds until  $1!$  is evaluated to be 1, which terminates the recursion. Figure 7.15(b) shows the values returned from each recursive call to its caller until the value is calculated and returned.



**Fig. 7.15** | Recursive evaluation of  $5!$ .

### 7.16.3 Implementing Factorial Recursively

Figure 7.16 uses recursion to calculate and display the factorials of the integers from 0 to 10. The recursive method `Factorial` (lines 17–28) first tests to determine whether a terminating condition (line 20) is true. If `number` is less than or equal to 1 (the base case), `Factorial` returns 1 and no further recursion is necessary. If `number` is greater than 1, line 26 expresses the problem as the product of `number` and a recursive call to `Factorial` evaluating the factorial of `number - 1`, which is a slightly simpler problem than the original calculation, `Factorial(number)`.

```

1 // Fig. 7.16: FactorialTest.cs
2 // Recursive Factorial method.
3 using System;

```

**Fig. 7.16** | Recursive `Factorial` method. (Part I of 2.)

```
4
5 class FactorialTest
6 {
7     static void Main()
8     {
9         // calculate the factorials of 0 through 10
10        for (long counter = 0; counter <= 10; ++counter)
11        {
12            Console.WriteLine($"{counter}! = {Factorial(counter)}");
13        }
14    }
15
16    // recursive declaration of method Factorial
17    static long Factorial(long number)
18    {
19        // base case
20        if (number <= 1)
21        {
22            return 1;
23        }
24        else // recursion step
25        {
26            return number * Factorial(number - 1);
27        }
28    }
29 }
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

**Fig. 7.16** | Recursive `Factorial` method. (Part 2 of 2.)

Method `Factorial` (lines 17–28) receives a parameter of type `long` and returns a result of type `long`. As you can see in Fig. 7.16, factorial values become large quickly. We chose type `long` (which can represent relatively large integers) so that the app could calculate factorials greater than  $20!$ . Unfortunately, the `Factorial` method produces large values so quickly that factorial values soon exceed even the maximum value that can be stored in a `long` variable. Due to the restrictions on the integral types, variables of type `float`, `double` or `decimal` might ultimately be needed to calculate factorials of larger numbers. This situation points to a weakness in some programming languages—the languages are *not easily extended* to handle the unique requirements of various apps. As you know, C# allows you to create new types. For example, you could create a type `HugeInteger` for arbitrarily large integers. This class would enable an app to calculate the factorials of larger numbers. In fact,



the .NET Framework's `BigInteger` type (from namespace `System.Numerics`) supports arbitrarily large integers.



### Common Programming Error 7.11

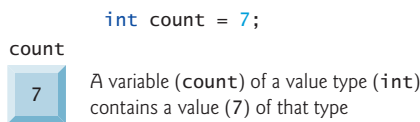
Either omitting the base case or writing the recursion step incorrectly so that it does not converge on the base case will cause *infinite recursion*, eventually exhausting memory. This error is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution.

## 7.17 Value Types vs. Reference Types

Types in C# are divided into two categories—*value types* and *reference types*.

### Value Types

C#'s simple types (like `int`, `double` and `decimal`) are all **value types**. A variable of a value type simply contains a *value* of that type. For example, Fig. 7.17 shows an `int` variable named `count` that contains the value 7.



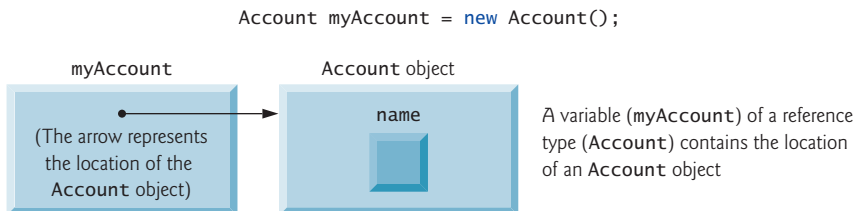
**Fig. 7.17** | Value-type variable.

### Reference Types

By contrast, a variable of a **reference type** (also called a **reference**) contains the *location* where the data referred to by that variable is stored. Such a variable is said to **refer to an object** in the program. For example, the statement

```
Account myAccount = new Account();
```

creates an object of our class `Account` (presented in Chapter 4), places it in memory and stores the object's reference in variable `myAccount` of type `Account`, as shown in Fig. 7.18. The `Account` object is shown with its name instance variable.



**Fig. 7.18** | Reference-type variable.

### Reference-Type Instance Variables Are Initialized to `null` by Default

Reference-type instance variables (such as `myAccount` in Fig. 7.18) are initialized by default to `null`. The type `string` is a reference type. For this reason, `string` instance variable `name`

is shown in Fig. 7.18 with an empty box representing the null-valued variable. A string variable with the value `null` is *not* an empty string, which is represented by "" or **string.Empty**. Rather, the value `null` represents a reference that does *not* refer to an object, whereas the empty string is a string object that does not contain any characters. In Section 7.18, we discuss value types and reference types in more detail.



#### Software Engineering Observation 7.4

*A variable's declared type (e.g., `int` or `Account`) indicates whether the variable is of a value type or a reference type. If a variable's type is one of the simple types (Appendix ), an enum type or a struct type (which we introduce in Section 10.13), then it's a value type. Classes like `Account` are reference types.*

## 7.18 Passing Arguments By Value and By Reference

Two ways to pass arguments to methods in many programming languages are **pass-by-value** and **pass-by-reference**. When an argument is passed by *value* (the default in C#), a *copy* of its value is made and passed to the called method. Changes to the copy do *not* affect the original variable's value in the caller. This prevents the accidental side effects that so greatly hinder the development of correct and reliable software systems. Each argument that's been passed in the programs so far has been passed by value. When an argument is passed by *reference*, the caller gives the method the ability to access and modify the caller's original variable—no copy is passed.

To pass an object by reference into a method, simply provide as an argument in the method call the variable that refers to the object. Then, in the method body, reference the object using the corresponding parameter name. The parameter refers to the original object in memory, so the called method can access the original object directly.

In the previous section, we began discussing the differences between *value types* and *reference types*. A major difference is that:

- *value-type variables store values*, so specifying a value-type variable in a method call passes a *copy* of that variable's value to the method, whereas
- *reference-type variables store references to objects*, so specifying a reference-type variable as an argument passes the method a *copy of the reference* that refers to the object.

Even though the reference itself is passed by value, the method can still use the reference it receives to interact with—and possibly modify—the original object. Similarly, when returning information from a method via a return statement, the method returns a copy of the value stored in a value-type variable or a copy of the reference stored in a reference-type variable. When a reference is returned, the calling method can use that reference to interact with the referenced object.



#### Performance Tip 7.1

*A disadvantage of pass-by-value is that, if a large data item is being passed, copying that data can take a considerable amount of execution time and memory space.*



#### Performance Tip 7.2

*Pass-by-reference improves performance by eliminating the pass-by-value overhead of copying large objects.*



### Software Engineering Observation 7.5

*Pass-by-reference can weaken security; the called method can corrupt the caller's data.*

#### 7.18.1 ref and out Parameters

What if you would like to pass a variable by reference so the called method can modify the variable's value in the caller? To do this, C# provides keywords **ref** and **out**.

##### *ref* Parameters

Applying the **ref** keyword to a parameter declaration allows you to pass a variable to a method by reference—the method will be able to modify the original variable in the caller. Keyword **ref** is used for variables that already have been initialized in the calling method.



### Common Programming Error 7.12

*When a method call contains an uninitialized variable as an argument to a **ref** parameter, the compiler generates an error.*

##### *out* Parameters

Preceding a parameter with keyword **out** creates an **output parameter**. This indicates to the compiler that the argument will be passed into the called method by reference and that the called method will assign a value to the original variable in the caller. This also prevents the compiler from generating an error message for an uninitialized variable that's passed as an argument to a method.



### Common Programming Error 7.13

*If the method does not assign a value to the **out** parameter in every possible path of execution, the compiler generates an error. Also, reading an **out** parameter before it's assigned a value is also a compilation error.*



### Software Engineering Observation 7.6

*A method can return only one value to its caller via a return statement, but can return many values by specifying multiple output (**ref** and/or **out**) parameters.*

#### *Passing Reference-Type Variables by Reference*

You also can pass a reference-type variable by reference, which allows you to modify it so that it refers to a new object. Passing a reference by reference is a tricky but powerful technique that we discuss in Section 8.13.



### Software Engineering Observation 7.7

*By default, C# does not allow you to choose whether to pass each argument by value or by reference. Value types are passed by value. Objects are not passed to methods; rather, references to objects are passed—the references themselves are passed by value. When a method receives a reference to an object, the method can manipulate the object directly, but the reference value cannot be changed to refer to a new object.*

### 7.18.2 Demonstrating ref, out and Value Parameters

The app in Fig. 7.19 uses the `ref` and `out` keywords to manipulate integer values. The class contains three methods that calculate the square of an integer.

---

```

1 // Fig. 7.19: ReferenceAndOutputParameters.cs
2 // Reference, output and value parameters.
3 using System;
4
5 class ReferenceAndOutputParameters
6 {
7     // call methods with reference, output and value parameters
8     static void Main()
9     {
10         int y = 5; // initialize y to 5
11         int z; // declares z, but does not initialize it
12
13         // display original values of y and z
14         Console.WriteLine($"Original value of y: {y}");
15         Console.WriteLine("Original value of z: uninitialized\n");
16
17         // pass y and z by reference
18         SquareRef(ref y); // must use keyword ref
19         SquareOut(out z); // must use keyword out
20
21         // display values of y and z after they're modified by
22         // methods SquareRef and SquareOut, respectively
23         Console.WriteLine($"Value of y after SquareRef: {y}");
24         Console.WriteLine($"Value of z after SquareOut: {z}\n");
25
26         // pass y and z by value
27         Square(y);
28         Square(z);
29
30         // display values of y and z after they're passed to method Square
31         // to demonstrate that arguments passed by value are not modified
32         Console.WriteLine($"Value of y after Square: {y}");
33         Console.WriteLine($"Value of z after Square: {z}");
34     }
35
36     // uses reference parameter x to modify caller's variable
37     static void SquareRef(ref int x)
38     {
39         x = x * x; // squares value of caller's variable
40     }
41
42     // uses output parameter x to assign a value
43     // to an uninitialized variable
44     static void SquareOut(out int x)
45     {
46         x = 6; // assigns a value to caller's variable
47         x = x * x; // squares value of caller's variable
48     }

```

---

**Fig. 7.19** | Reference, output and value parameters. (Part I of 2.)

```

49
50 // parameter x receives a copy of the value passed as an argument,
51 // so this method cannot modify the caller's variable
52 static void Square(int x)
53 {
54     x = x * x;
55 }
56 }

```

```

Original value of y: 5
Original value of z: uninitialized

Value of y after SquareRef: 25
Value of z after SquareOut: 36

Value of y after Square: 25
Value of z after Square: 36

```

**Fig. 7.19** | Reference, output and value parameters. (Part 2 of 2.)

Method `SquareRef` (lines 37–40) multiplies its parameter `x` by itself and assigns the new value to `x`. `SquareRef`'s parameter is declared as `ref int`, which indicates that the argument passed to this method must be an integer that's passed by reference. Because the argument is passed by reference, the assignment at line 39 modifies the original argument's value in the caller.

Method `SquareOut` (lines 44–48) assigns its parameter the value 6 (line 46), then squares that value. `SquareOut`'s parameter is declared as `out int`, which indicates that the argument passed to this method must be an integer that's passed by reference and that the argument does *not* need to be initialized in advance.

Method `Square` (lines 52–55) multiplies its parameter `x` by itself and assigns the new value to `x`. When this method is called, a *copy* of the argument is passed to the parameter `x`. Thus, even though parameter `x` is modified in the method, the original value in the caller is *not* modified.

Method `Main` (lines 8–34) invokes methods `SquareRef`, `SquareOut` and `Square`. We begin by initializing variable `y` to 5 and declaring, but *not* initializing, variable `z`. Lines 18–19 call methods `SquareRef` and `SquareOut`. Notice that when you pass a variable to a method with a reference parameter, you must precede the argument with the same keyword (`ref` or `out`) that was used to declare the reference parameter. Lines 23–24 display the values of `y` and `z` after the calls to `SquareRef` and `SquareOut`. Notice that `y` has been changed to 25 and `z` has been set to 36.

Lines 27–28 call method `Square` with `y` and `z` as arguments. In this case, both variables are passed by *value*—only *copies* of their values are passed to `Square`. As a result, the values of `y` and `z` remain 25 and 36, respectively. Lines 32–33 output the values of `y` and `z` to show that they were *not* modified.



#### Common Programming Error 7.14

The `ref` and `out` arguments in a method call must match the `ref` and `out` parameters specified in the method declaration; otherwise, a compilation error occurs.

## 7.19 Wrap-Up

In this chapter, we discussed the difference between `non-static` and `static` methods, and we showed how to call `static` methods by preceding the method name with the name of the class in which it appears and the member-access operator (`.`). You saw that the `Math` class in the .NET Framework Class Library provides many `static` methods to perform mathematical calculations. We also discussed `static` class members and why method `Main` is declared `static`.

We presented several commonly used Framework Class Library namespaces. You learned how to use operator `+` to perform `string` concatenations. You also learned how to declare constants with the `const` keyword and how to define sets of named constants with `enum` types. We demonstrated simulation techniques and used class `Random` to generate sets of random numbers. We discussed the scope of fields and local variables in a class. You saw how to overload methods in a class by providing methods with the same name but different signatures. You learned how to use optional and named parameters.

We showed the concise notation of C# 6's expression-bodied methods and read-only properties for implementing methods and read-only property `get` accessors that contain only a `return` statement. We discussed how recursive methods call themselves, breaking larger problems into smaller subproblems until eventually the original problem is solved. You learned the differences between value types and reference types with respect to how they're passed to methods, and how to use the `ref` and `out` keywords to pass arguments by reference.

In Chapter 8, you'll maintain lists and tables of data in arrays. You'll see a more elegant implementation of the app that rolls a die 60,000,000 times and two versions of a `GradeBook` case study. You'll also access an app's command-line arguments that are passed to method `Main` when a console app begins execution.

# Index

## Symbols

$\wedge$ , boolean logical exclusive OR 143, **145**  
truth table 146  
--, prefix/postfix decrement 118, 119  
-, subtraction 60, 61  
!, logical negation 143, **146**  
truth table 146  
!=, not equals 62  
?:, ternary conditional operator 103, 120  
?? (null coalescing operator) 391, **392**  
?., null-conditional operator (C# 6) **390**, 391, 607  
?[], null-conditional operator (C# 6) **607**  
. (member access operator) **70**  
.NET Core xxxv  
"", empty string **190**  
{, left brace 45  
, right brace 45  
@ verbatim string character 504  
\*, multiplication 60, 61  
\*=:, multiplication compound assignment operator 118  
/ forward slash in end tags **692**  
/, division 60, 61  
/\* \*/ delimited comment 42  
//, single-line comment 42  
\\, escape character 55  
\", double-quote escape sequence 54  
\\=, division compound assignment operator 118  
\\n, newline escape sequence 54  
\\r, carriage-return escape sequence 55  
\\t, horizontal tab escape sequence 54

&, boolean logical AND 143, **145**  
&, menu access shortcut 440, 442  
&&, conditional AND **143**, 144  
truth table 144  
%, remainder 60, 61  
%=:, remainder compound assignment operator 118  
+, addition 60, 61  
+, concatenation operator 515  
++, prefix/postfix increment 118  
+=, addition compound assignment operator 117  
<, less than 62  
<=, less than or equal 62  
<>, angle brackets for XML elements **692**  
=:, assignment operator 58  
-=:, subtraction compound assignment operator 117  
==, comparison operator 508  
==, is equal to 62  
=> (in an expression-bodied method) **185**  
=>, lambda operator **613**, 650  
>, greater than 62  
>=, greater than or equal to 62  
|, boolean logical inclusive OR 143, **145**  
||, conditional OR 143, **144**  
truth table **144**  
\$(dollar sign for interpolated string) **56**

## A

Abs method of Math 153  
absolute value 153  
abstract class **334**, 335, 353  
abstract keyword 316, **334**  
abstract method **334**, 336, 338

abstraction 71  
AcceptButton property of class Form 398  
AcceptsReturn property of class TextBox 411  
access modifier **75**  
private **75**, 270, 303  
protected 270, **303**  
public **75**, 270, 303  
access modifier in the UML  
- (private) 76  
+ (public) 76  
access private member of a class 76  
access shortcut 439  
action 100, 106  
action expression in the UML **98**  
action state in the UML **98**  
action state symbol **97**  
Activation property of class ListView **474**  
activation record **177**  
active control **407**  
active tab **21**  
active window **397**  
ActiveLinkColor property of class LinkLabel 453  
ActiveMdiChild property of class Form 485, 486  
activity diagram 98, 100  
do...while statement 133  
for statement 128  
if statement 100  
if...else statement 101  
in the UML 106  
sequence statement 97  
switch statement 138  
while statement 107  
activity in the UML **97**  
add a database to a project 639

- add a reference to a class library 532, 644
- Add method of class `List<T>` **260**
- Add method of class
  - `ObjectCollection` 459
- Add method of class
  - `SortedDictionary<K, V>` **602**
- Add Tab menu item 480
- Add User Control... option in Visual Studio .NET 499
- Add Windows Form... option in Visual Studio 485
- AddDay method of structure
  - `DateTime` **453**
- addition 60
- AddLast method of class
  - `LinkedList` **605**
- AddRange method of class
  - `List<T>` 257
- AddYears method of structure
  - `DateTime` **453**
- ADO.NET Entity Data Model **636**
  - data source for data binding 645
  - entities 642
- ADO.NET Entity Framework **631**
  - `DbContext` class 649
  - `DbExtensions` class **650**
  - Entity Data Model **636**
- AfterSelected event of class
  - `TreeView` 469
- Aggregate LINQ extension method **619**
- algebraic notation 60
- Alphabetic icon 27
- alphabetizing 507
- Alt* key 433
- Alt* key shortcut 439
- Alt property of class
  - `EventArgs` 434, 436
- Analyze menu 21
- anchor a control **407**
- Anchor property of class `Control` 409
- anchoring a control 407
- Anchoring demonstration 408
- Android
  - operating system 9
- angle bracket (<>) for XML elements **692**
- anonymous method **590, 611, 650**
- anonymous type **256, 659**
  - `Equals` method 659
  - `ToString` method 659
- Any extension method of interface `IEnumerable<T>` **255**
- app 45
- app bar 9
- App Developer Agreement 10
- Appearance property of class
  - `CheckBox` 416
- Append method of class
  - `StringBuilder` 520
- AppendFormat method of class
  - `StringBuilder` 521, 522
- AppendText method of class `File` 557
- application 16
- Application class **448**
  - `Exit` method **448, 461**
- arbitrary number of arguments 236
- args parameter of `Main` method 238
- argument promotion **160**
- argument to a method **46**
- ArgumentException class **602**
- ArgumentOutOfRangeException class **268, 276, 505, 514, 522**
- arithmetic compound
  - assignment operators **117**
- arithmetic operators 59
- ArrangeIcons value of enumeration `MailLayout` 487
- array
  - bounds checking **209**
  - ignoring element zero 210
  - Length property **198**
  - pass an array element to a method 217
  - pass an array to a method 217
- array-access expression **197**
- Array class 244, 592, 593, 596
  - Resize method **198**
- Array class static methods for common array manipulations 593
- array-creation expression **198, 199**
- array initializer **200**
  - for jagged array 226
  - for rectangular array 225
  - nested **225**
- array-access expression
  - for jagged arrays 226
  - for rectangular arrays 225
- ArrayList class 592
- arrays as references 240
- arrow 98
- as operator (downcasting) 349, 391
- ascending modifier of a LINQ orderby clause **250**
- ASCII (American Standard Code for Information Interchange) 436
  - character set 138, 704
- ASP.NET **7**
- AsParallel method of class
  - `ParallelEnumerable` **625**
- assembly (compiled code) 51, 644
- assign a value to a variable 58
- assignment operator
  - = **58, 61**
  - compound **117**
- assignment statement **58**
- associativity of operators **61, 65, 120**
  - left to right 65, 121
  - right to left 61, 65, 120
- async xxv, 7
- async modifier 674, **675, 678**
- asynchronous programming **7, 674**
- asynchronous task **673, 675**
- attribute
  - in the UML 4, 76
  - of a class 2
  - of an object 4
- Attribute method of class
  - `XElement` **693**
- attributes (XML) **692**
- augmented reality 10



- AuthorISBN table of Books
    - database 632, 634
  - Authors table of Books database
    - 632, 633
  - auto-implemented property **83**,
    - 87, 534
    - getter only (C# 6) **212**, 282
  - auto-hide 24
  - autoincremented database
    - column **633**
  - automatic garbage collection
    - 374
  - automatic memory
    - management 284
  - AutoPopDelay property of class
    - ToolTip 427
  - AutoScroll property of class
    - Form 398
  - AutoScroll property of class
    - Panel 413
  - AutoSize property of class
    - TextBox **33**
  - average calculation 107, 109,
    - 110
  - Average IEnumerable<T>
    - extension method **619**, 625
  - Average ParallelQuery<T>
    - extension method 625
  - await xxv
  - await expression 7, **675**, 679
  - await multiple Tasks 685
  - awaitable entity 675
- B**
- BackColor property of a form **32**
  - BackColor property of class
    - Control 406
  - background color 32
  - BackgroundImage property of
    - class Control 406
  - backslash, (\) **54**
  - bar chart 205, 206
  - bar of asterisks 205, 206
  - base
    - for constructor initializers **316**
    - for invoking overridden methods 323
    - keyword 303, 316, 323, 324
  - base case 186
  - base class **300**
    - constructor 307
    - default constructor 307
    - direct **300**, 302
    - indirect **300**, 302
    - method overridden in a
      - derived class 323
  - behavior of a class 2
  - BigInteger struct 189
  - binary operator **58**, 59, 146
  - binary search tree 599
  - BinaryFormatter class **550**
    - Deserialize method **550**
    - Serialize method 554
  - BinarySearch method of class
    - Array **596**
  - BindingNavigator class **638**, 647
  - BindingSource class **647**
    - DataSource property **650**
    - EndEdit method **651**
    - MoveFirst method **654**
    - Position property **654**
  - BitArray class 593
  - bitwise operators 418
  - bitwise XOR operator 449
  - blank line 43
  - block of statements **64**, **102**,
    - 113, 159
  - BMP (Windows bitmap) 36
  - body
    - of a class declaration 45
    - of a method **46**
    - of an if statement 61
  - body of a loop 106
  - Bohm, C. 97
  - Books database 632
    - table relationships 635
  - bool simple type **99**, 702
    - expression 103
  - Boolean 99
  - boolean logical AND, & 143,
    - 145**
  - boolean logical exclusive OR, ^
    - 143, **145**
    - truth table 146
  - boolean logical inclusive OR, |
    - 145**
  - BorderStyle property of class
    - Panel 413
  - boundary of control 498
  - bounds checking **209**
  - boxing 592
  - braces { and } 102, 113
  - braces {} 64
  - braces not required 137
  - braces, { } 200
  - break statement **136**, 141
    - exiting a for statement 142
  - brittle software **320**
  - buffer **531**
  - BufferedStream class **531**
  - buffering **531**
  - Build menu 21
  - built-in array capabilities 592
  - button 396
  - Button class 12, 396, 410
    - Click event 411
    - FlatStyle property 411
    - Text property 411
  - Button properties and events
    - 411
  - Button property of class
    - MouseEventArgs 431
  - ButtonBase class **410**
  - byte simple type 702
- C**
- C format specifier 92
  - C format specifier (for currency)
    - 91**
  - C# 6 **185**
    - Add extension method
      - support in collection initializers 608
    - exception filter **392**
    - expression-bodied method **185**
    - expression-bodied property **185**
    - getter-only auto-implemented property **212**, 282, 289
    - index initializer 608
    - nameof operator **276**
    - null-conditional operator
      - (?.) **390**, 391, 607
    - null-conditional operator
      - (?[]) **607**
    - string interpolation **55**, 56
    - using static **595**

- C# 6 Specification xxi, 373
- C# Coding Conventions 207
- .cs file name extension 45
- C# keywords 44
- C# programming language **5**
- Calculating values to be placed into the elements of an array 201
- calculations 66, 97
- CalendarForeColor property of class DateTimePicker 450
- CalendarMonthBackground property of class DateTimePicker 450
- call stack 382
- callback method 675
- calling method (caller) 73
- camel case **44**, 58, 72
- CancelButton property of class Form 398
- CancelPendingRequests method of class HttpClient **691**
- Capacity property of class List<T> **257**
- Capacity property of class StringBuilder 518
- Card class represents a playing card 212
- card games 212
- card shuffling
  - Fisher-Yates 215
- Card shuffling and dealing application 216
- carriage return 55
- Cascade value of enumeration MdiLayout 487
- cascaded method calls **297**
- cascaded window **487**
- case **136**, 137
  - keyword **136**
- case sensitive **44**
- casino 164, 169
- cast
  - downcast **349**
- cast operator **112**, 113, 161, 173
- catch
  - general catch clause **370**
  - catch all exception types 370
  - catch an exception **367**
  - Catch block **211**
  - catch block **369**
    - when clause (C# 6) 392
    - with no exception type 370
    - with no identifier 369
  - catch-related errors 373
  - Categorized** icon **27**
  - Ceiling method of Math 153
  - char simple type 57, 702
  - array 505
  - Char struct 503
    - CompareTo method 527
    - IsDigit method 526
    - IsLetter method 527
    - IsLetterOrDigit method 527
    - IsLower method 527
    - IsPunctuation method 527
    - IsSymbol method 527
    - IsUpper method 527
    - IsWhiteSpace method 527
    - static character-testing methods and case-conversion methods 525
    - ToLower method 527
    - ToUpper method 527
  - character 163
    - constant **138**, **504**
    - string **46**
  - check box **410**
  - CheckBox class 396, **416**
    - Appearance property 416
    - Checked property 416
    - CheckedChanged event 416
    - CheckedState property 416
    - properties and events 416
    - Text property 416
    - ThreeState property **416**
  - CheckBoxes property
    - of class ListView **474**
    - of class TreeView 469
  - Checked property
    - of class CheckBox 416
    - of class RadioButton 419
    - of class ToolStripMenuItem 443, **448**
    - of class TreeNode 469
  - CheckedChanged event
    - of class CheckBox 416
    - of class RadioButton 419
  - CheckedIndices property of class CheckedListBox 462
  - CheckedItems property of class CheckedListBox 462
  - CheckedListBox class 439, **457**, 461
    - CheckedIndices property 462
    - CheckedItems property 462
    - GetItemChecked method 462
    - ItemCheck event **461**, 462
    - properties and events 462
    - SelectionMode property 462
  - OnClick property of class ToolStripMenuItem 443
  - CheckedState property of class CheckBox 416
  - child node **468**
  - child window **484**
    - maximized 486
    - minimized 486
  - Choose Items...** option in Visual Studio 500
  - chromeless window 9
  - class **2**, **3**, 152
    - constructor 84
    - declaration 44, 45
    - default constructor **86**
    - instance variable **154**
    - name **44**, 45, 493
    - user defined **44**
  - class average 107
  - class cannot extend a sealed class 351
  - class constraint **576**
  - class hierarchy **300**, 335
  - class library **7**, **301**, 325, 493
    - add a reference 644
    - add a reference to 532
    - compile into 532
  - class variable **154**
  - Class View** (Visual Studio .NET) 289
  - “class-wide” information **285**
  - Classes
    - Application **448**
    - ArgumentException 602
    - ArgumentOutOfRangeException 276
    - Array 244, 592, 593, 596, 597

## Classes (cont.)

ArrayList 592  
 BinaryFormatter **550**  
 BindingNavigator **647**  
 BindingSource **647, 651**  
 BitArray 593  
 BufferedStream **531**  
 Button 410  
 ButtonBase **410**  
 CheckBox **416**  
 CheckedListBox 439, **457**,  
 461  
 ComboBox 439, **464**  
 Console **46**, 53, 530, 531  
 Control **406**, 409, 497  
 DataContractJson-  
 Serializer 550  
 DataGridView **638**  
 DateTimePicker **450**  
 DbContext **637**, 643  
 Delegate **404**  
 Dictionary **565**, 592  
 Dictionary<K, V> 598  
 Directory **557**, 561  
 DirectoryInfo **479, 557**  
 DivideByZeroException  
**365**, 368, 372  
 Enumerable 616, 637  
 EventArgs 400  
 Exception **372**  
 ExecutionEngineException  
**372**  
 File **557**, 561, 565  
 FileInfo **479**  
 FileStream **531**  
 Font **418**  
 Form 397, 398, 485, 486  
 FormatException **366**, 369  
 Graphics **433**, **467**  
 GroupBox **413**  
 Hashtable **602**  
 HttpClient **691**  
 ImageList **469**, 475  
 IndexOutOfRangeException  
 211  
 InvalidCastException 349,  
 391, 591  
 InvalidOperationException  
 597, 606

## Classes (cont.)

ItemCheckEventArgs 462  
 KeyEventArgs **433**, 434, 436  
 KeyNotFoundException 602  
 LinkedList 592, **605**  
 LinkedList<T> 592, **603**  
 LinkedListNode<T> **603**  
 LinkLabel 439, **453**, 453  
 List 592  
 List<T> **256**, 257, **260**, 592  
 ListBox 439, **456**  
 ListBox.ObjectCollection  
 458  
 ListView **474**  
 ListViewItem **475**  
 Match 503, 527  
 Math 153  
 MemoryStream **531**  
 MenuStrip **440**  
 MonthCalendar **449**  
 MouseEventArgs **430**  
 MulticastDelegate **404**  
 NullReferenceException  
**372**  
 NumericUpDown 396, **428**  
 object **305**, 325  
 ObjectCollection **458**, **459**,  
**461**  
 ObservableCollection<T>  
**650**, 655  
 OpenFileDialog **543**, **549**  
 OutOfMemoryException **372**  
 PaintEventArgs **497**  
 Panel **413**  
 ParallelEnumerable **625**  
 ParallelQuery<T> **625**  
 Path **473**, **565**  
 PictureBox 424, 487  
 Process **456**  
 ProgressBar **694**  
 Queryable **637**  
 Queue<T> **592**, 593  
 RadioButton **416**, 419  
 Random 164  
 ResourceManager **426**  
 Resources **426**  
 SaveFileDialog 538  
 SolidBrush **433**  
 SortedDictionary 592, **599**,  
 601

## Classes (cont.)

SortedDictionary<K, V> 592  
 SortedList 592, 593  
 SortedList<K, V> **592**  
 SortedSet<T> **627**  
 Stack 577  
 Stack<T> **592**, 593  
 StackOverflowException  
**372**  
 Stream 531, **531**  
 StreamReader **531**  
 StreamWriter **531**  
 string 73, 503  
 StringBuilder 503, 517,  
 520, 521, 522  
 SystemException **372**  
 TabControl **480**  
 TabPage **480**  
 Task<TResult> **679**  
 TextBox 396  
 TextReader 531  
 TextWriter 531  
 Timer **499**  
 ToolStripMenuItem **440**,  
**442**  
 ToolTip **426**, 427  
 TreeNode **469**  
 TreeNodeCollection **469**  
 TreeView 439, **468**, 469  
 TreeViewEventArgs **469**  
 Type 326, **350**  
 UnauthorizedAccess-  
 Exception 473  
 UserControl **497**  
 ValueType **525**  
 XAttribute **693**  
 XDocument **693**  
 XElement **693**  
 XmlSerializer 550  
 Clear method of class Array **597**  
 Clear method of class  
 Dictionary **565**  
 Clear method of class Graphics  
**467**  
 Clear method of class List<T>  
 257  
 Clear method of class  
 ObjectCollection 461  
 ClearSelected method of class  
 ListBox 458

- click a Button 398, 410
- Click event of class Button 411
- Click event of class PictureBox 424
- Click event of class ToolStripMenuItem 442, 443
- Clicks property of class MouseEventArgs 431
- client of a class 73, 78
- client code 330
- client of a class 75
- ClipRectangle property of class PaintEventArgs 497, 498
- clock 498
- cloning objects
  - shallow copy **326**
- close a project 21
- close a window 398
- close box 38
- Close method of class Form 398
- CLR (Common Language Runtime) 7, 374, 388
- code reuse 300, 589
- code snippets **174**
- Coding Conventions (C#) 207
- coding requirements 207
- coin tossing 165
- collapse a tree 25
- Collapse method of class TreeNode 470
- collapse node 469
- collection 256, 569, **589**
- collection class **589**
- collection initializer **263**, 608
  - Add extension method (C#) 608
- collision **598**
- Color structure **433**
- column 225
- column in a database table **631**, 632
- column index **229**
- columns of a two-dimensional array **225**
- ComboBox class 396, 439, **464**
  - DropDownStyle property **464**, 465
  - Items property **465**
  - MaxDropDownItems property **464**
- ComboBox class (cont.)
  - SelectedIndex property **465**
  - SelectedIndexChanged event **465**
  - SelectedIndexChanged event handler 654
  - SelectedItem property **465**
  - Sorted property 465
- ComboBox demonstration 464
- ComboBox properties and an event 465
- ComboBox used to draw a selected shape 465
- ComboBoxStyle enumeration **464**
  - DropDown value 464
  - DropDownList value 464
  - Simple value 464
- comma (,) 129
- comma-separated list 129
  - of parameters 157
  - of arguments 74
- command-line argument **237**, 239
- Command Prompt **41**, 238
- comment 42
- CommissionEmployee class 305, 320
  - extends Employee 343
- Common Language Runtime (CLR) 7, 374, 388
- Common Programming Errors overview xxvii
- CompareTo method
  - of interface IComparable 360, 527, **574**
- comparison operator **61**, 360
- compartment in a UML class diagram 76
- compilation error **42**
- compile 47
- compile into a class library 532
- compile-time error **42**
- compiler 113
- compiler error **42**
- compile-time type safety **568**
- ComplexNumber class 292
- component 2, **396**
- component tray **427**, 647
- composite key **631**
- composite primary key 634
- composition **280**, 301, 303
- compound assignment operators **117**, 120
  - \*= 118
  - \= 118
  - %= 118
  - += 117
  - = 117
- compound interest 129
  - calculating with for 129
- Concat method of class string 515
- concatenate strings 286
- concrete class **334**
- concrete derived class 339
- concurrent operations 673
- condition 61, 133
- conditional AND (&&) operator **143**, 145, 255
  - truth table 144
- conditional expression **103**
- conditional operator, ?: 103, 120
- conditional OR, || 143, **144**
  - truth table **144**
- confusing the equality operator == with the assignment operator = 61
- connect to a database 637, 639
- connection string 641, 645
- console app **41**, 47
- Console class 530, 531
- console window 41, 53, 54
- Console.WriteLine **46**, 53
- const keyword 139, **154**, 202, 288
- constant **139**, 154, 202
  - declare 202
  - must be initialized 202
  - Pascal Case 202
- constant integral expression **133**, 138
- constant string expression **133**
- Constants
  - Nan of structure Double **366**, 388
  - NegativeInfinity of structure Double **366**
  - PositiveInfinity of structure Double **366**

- constituent controls **497**
- constructor 84
  - multiple parameters 87
- constructor constraint (`new()`) **576**
- constructor header 85
- constructor initializer **276**, 316
  - with keyword `base` 316
- constructors cannot specify a return type 85
- container 396, **397**
  - parent **409**
- container control in a GUI **407**
- Contains method of class `List<T>` 257, **260**
- ContainsKey method of class `Dictionary` **565**
- ContainsKey method of `SortedDictionary<K,V>` **602**
- context-sensitive help **28**
- contextual keyword 82
  - value **82**
- contextual keywords 44
- continue keyword **141**
- continue statement 141, 142
  - terminating an iteration of a for statement 143
- contravariance **627**
- control **20**, 26, 396
- control boundary 498
- Control class **406**, 497
  - Anchor property 409
  - BackColor property 406
  - BackgroundImage property 406
  - Dock property 409
  - Enabled property **407**
  - Focused property 406
  - Font property 407
  - ForeColor property 407
  - Hide method 407
  - KeyDown event **433**, 434
  - KeyPress event **433**, 434
  - KeyUp event **433**, 434
  - Location property 409
  - MaximumSize property 409
  - MinimumSize property 409
  - MouseDown event 431
  - MouseEnter event 431
  - MouseHover event 431
- Control class (cont.)
  - MouseLeave event 431
  - MouseMove event 431
  - MouseUp event 431
  - MouseWheel event 431
  - OnPaint method **497**
  - Padding property 409
  - Select method **407**
  - Show method 407
  - Size property 409
  - TabIndex property **407**
  - TabStop property 407
  - Text property 407
  - Visible property 407
- control layout and properties 406
- Control property of class `KeyEventArgs` 434, 436
- control statement 97, 99, 100
  - nesting **99**
  - stacking **99**
- control variable **124**, 126, 127
- Controls **12**
  - BindingNavigator **647**
  - Button 12
  - DataGridView **638**
  - GroupBox 12
  - Label **20**, **29**, 32
  - Panel 12
  - PictureBox **20**, **29**, 35
  - RadioButton 12
- Controls property of class `GroupBox` 413, **414**
- Controls property of class `Panel` 413
- converge on a base case 186
- convert
  - an integral value to a floating-point value 162
- Copy method of class `Array` **596**
- Copy method of class `File` 557
- copying objects
  - shallow copy **326**
- CopyTo method of class `string` 506
- Cos method of `Math` 153
- cosine 153
- Count extension method of interface `IEnumerable<T>` **255**
- Count method (LINQ) 565
- Count property
  - of class `List<T>` **257**
  - of `SortedDictionary<K,V>` **602**
- counter-controlled iteration **107**, 112, 115, 124, 125
  - with the for iteration statement 125
  - with the while iteration statement 125
- counting loop 125
- covariance **626**
- covariant
  - interface 626
- craps (casino game) 164, 169
- create a reusable class 493
- create an object (instance) of a class 69, 70
- Create method of class `File` 557
- CreateDirectory method of class `Directory` **557**
- CreateText method of class `File` 557
- creating a child Form to be added to an MDI Form 485
- creating a generic method 584
- creating and initializing an array 199
- credit inquiry 545
- .cs file name extension 25, 72
- .csproj file extension 36
- Ctrl key 136, 433
- Ctrl + z 136
- culture settings 91, 114
- Current property of `IEnumerator` **596**
- current time 499
- CurrentValue property of class `ItemCheckEventArgs` 462
- cursor 46, 53
- custom control **497**, 498
  - creation 498, 500
- Custom palette 32
- Custom tab **32**
- Custom value of enumeration `DateTimePickerFormat` **450**
- CustomFormat property of class `DateTimePicker` **450**
- customize a Form 26
- customize Visual Studio IDE 21

## D

- D format specifier for integer values 92, 206
- data binding **637**
- data source 249, 615
  - entity data model 645
- Data Source Configuration Wizard **645**
- Data Sources window **645**, 646
- data types
  - bool **99**
  - double **110**
  - float **110**
- database **630**
  - add to a project 639
  - saving changes in LINQ to Entities 650
  - schema **632**
- database connection 639
- database management system (DBMS) **630**
- database schema **632**, 636
- database table **631**
- DataContext class
  - SaveChanges method **637**
- DataContractJsonSerializer class 550
- DataGridView control **638**, 638, 646
- DataSource property
  - BindingSource class **650**
- Date property of a DateTime **450**
- DateChanged event of class
  - MonthCalendar **449**, 450
- DateTime structure **499**, 680
  - AddDay method **453**
  - AddYears method **453**
  - DayOfWeek property **453**
  - Now property **499**, **625**
  - Subtract method **625**
  - ToLongDateString method **453**
  - ToLongTimeString method **499**
- DateTimePicker class **450**
  - CalendarForeColor property 450
  - CalendarMonthBackground property 450
  - CustomFormat property **450**
  - DateTimePicker class (cont.)
    - Format property **450**
    - MaxDate property 450, **453**
    - MinDate property 450, **453**
    - ShowCheckBox property 450
    - ShowUpDown property 451
    - Value property **450**, 451, 452
    - ValueChanged event **450**
- DateTimePickerFormat enumeration **450**
  - Custom value 450
  - Long value 450
  - Short value 450
  - Time value 450
- DayOfWeek enumeration **453**
  - property of structure
    - DateTime **453**
- DB2 630
- DbContext class **637**, 643, 649
  - SaveChanges method 651
- DbExtensions class **650**
  - Load extension method **650**, 654
- DBMS (database management system) **630**
- dealing a card 212
- Debug menu 21
- Debugging 21
- decimal literal **89**
- decimal point 110, 114
- decimal simple type 57, **87**, 130, 703
  - Parse method **92**
- decimal type
  - Parse method 93
- DecimalPlaces property of class
  - NumericUpDown **428**
- decision **61**, 99
- decision symbol **99**
- declaration **55**
  - class 44, **45**
  - method **46**
- declarative programming **247**
- declare a constant 202
- decrement operator, -- 118, 119
- default
  - case in a switch 136
  - keyword 136
- default case 168
- default constructor **86**, 279, 307
- default event of a control **404**
- default settings 12
- default type constraint (object)
  - of a type parameter 579
- default value **73**, 121
- default value for optional parameter **183**, 183
- deferred execution **263**, 621
- definitely assigned **109**, 172
- Delegate 686
- delegate **403**, 590, 608
  - Delegate class **404**
  - Func **619**, 621, 649, 650, 655
  - MulticastDelegate class **404**
  - registering an event handler **403**
- delegate keyword **403**, 610
- Delete method of class
  - Directory 558
- Delete method of class File
  - 557, **565**
- delimited comments **42**
- dependent condition 145
- derived class **300**
- Descendants method of class
  - XDocument **693**
- descending modifier of a LINQ
  - orderby clause **250**
- deselected state **419**
- Deserialize method of class
  - BinaryFormatter **550**
- deserialized object **550**
- design mode 38
- design process **5**
- Design view **19**, 30
- destructor 284
- dialog **19**
- DialogResult enumeration **424**, **538**
- diamond 99
  - in the UML **97**
- dice game 169
- dictionary **597**
- Dictionary class 592
- Dictionary<K,V> class **565**, 592, 598
  - Clear method **565**

- Dictionary<K,V> class (cont.)
  - ContainsKey method **565**
  - Keys property **565**
  - Remove method **565**
- digit 57
- direct base class 300, 302
- Directory class **557**, 561
  - CreateDirectory method **557**
  - GetFiles method **565**
  - methods (partial list) 558
- DirectoryInfo class **479**, **557**
  - Exists method **479**
  - FullName property **479**
  - GetDirectories method **479**
  - GetFiles method **479**
  - Name property **479**
  - Parent property **479**
- display output 65
- displaying line numbers in the IDE xxxv
- Dispose method of interface
  - IDisposable 361, 381
- distance between values
  - (random numbers) 168
- Distinct extension method of
  - interface IEnumerable<T> **255**
- divide and conquer 186
- divide by zero 365, 368
- DivideByZeroException class
  - 365**, 368, 369, 372
- division 60
- .dll file **51**, **494**
- do keyword **132**
- do...while iteration statement
  - 98, 132, 133
- dock a control **407**
- Dock property of class Control
  - 409, 647
- docking demonstration 409
- .NET 8
  - Framework 7, 569
  - Framework Class Library (FCL) **5**, **8**, **43**, 152, 162, 360
  - Framework documentation **43**
  - initiative 7
  - .NET 4.6 8
  - .NET Core **xxxv**, **6**, **8**
  - dotted line in the UML 98
  - Double 574
    - (double) cast **113**
  - double data type **110**
  - double equals, == 61
  - double quote, " 46, 54
  - double-selection statement 98
  - double simple type **57**, 131, 703
    - Parse method **157**
  - Double.NaN **366**, 388
  - Double.NegativeInfinity **366**
  - Double.PositiveInfinity **366**
  - down-arrow button 32
  - downcast **349**
  - drag the mouse **27**
  - Draw event of class ToolTip 427
  - draw on control 498
  - DrawEllipse method of class
    - Graphics **468**
  - DrawPie method of class
    - Graphics **468**
  - DrawRectangle method of class
    - Graphics **468**
  - DreamSpark xxviii
  - driver class **69**
  - drop-down list 396, **464**
  - DropDown value of enumeration
    - ComboBoxStyle 464
  - DropDownList value of
    - enumeration ComboBoxStyle 464
  - DropDownStyle property of class
    - ComboBox **464**, 465
  - dummy value **110**
  - DVD 530
  - dynamic binding **348**
  - dynamic resizing **246**
  - dynamic resizing of a List
    - collection 257
  - dynamically linked library **51**, **494**

## E

  - E format specifier 92
  - ECMA-334 (C# Standard) 6
  - Edit menu 21
  - editable list 465
  - EF (ADO.NET Entity Framework) **631**
  - element (XML) **692**
  - element of an array **197**
  - element of chance 164
  - eligible for destruction **284**, 287
  - eligible for garbage collection **284**, 287
  - eliminate resource leak 375
  - ellipsis button **33**
  - else 100
  - Employee abstract base class 338, 357
  - Employee class with FirstName, LastName and MonthlySalary properties 251
  - Employee class with references to other objects 283
  - Employee hierarchy test application 346
  - empty parameter list 75
  - empty statement (a semicolon, ;) 103
  - empty string 190
    - "" **190**
    - string.Empty **190**
  - EmptyStackException indicates a stack is empty 580
  - Enabled property of class
    - Control **407**
  - encapsulation **4**, 312, 351
  - “end of data entry” 110
  - end-of-file indicator **530**
    - EOF **136**
  - end tag **692**
  - EndEdit method of class
    - BindingSource **651**
  - EndsWith method of class string
    - 510, 511
  - EnsureCapacity method of class
    - StringBuilder 518
  - Enter* (or *Return*) key 30, 46
  - enter data from the keyboard 396
  - entities in an entity data model
    - 642
  - entity connection string 641
  - Entity Data Model 631, **636**, 638, 649
    - ADO.NET Entity Framework **636**
    - create from database 639
    - data source for data binding 645
    - entities 642

- entity-relationship diagram **635**
  - entry point 46
    - of an application **154**
  - enum **172**
    - keyword 172
  - Enumerable class 616, 637
    - Range method **625**
    - Repeat method **697**
    - ToArray method 263
    - ToList method 263, **625**
  - enumeration **172**
  - enumeration constant **173**
  - enumerations
    - ComboBoxStyle **464**
    - DateTimePickerFormat **450**
    - DayOfWeek **453**
    - MdiLayout **487**
    - SelectionMode **457**
  - enumerator **589**, 596
    - fail fast 597
    - of a LinkedList 606
  - equal likelihood 166
  - equality operators (== and !=) 99
  - Equals method of an
    - anonymous type 659
  - Equals method of class object 325
  - Equals method of class string 508, 509
  - Error List** window **51**
  - Error property of class Console 530
  - escape character **54**
  - escape sequence **54**, 57
    - carriage return, \r 55
    - escape character, \ 55
    - horizontal tab, \t 54
    - newline, \n 54, 57
  - event **398**
  - event driven **6**, **398**
  - event handler **398**, 403
  - event handling **398**
  - event handling model 398
  - event multicasting **404**
  - event sender **403**
  - EventArgs class 400
  - events **6**
  - events at an interval 499
  - exception 58, 209, **211**, **363**
    - ArgumentException 602
    - exception (cont.)
      - handler **211**
      - handling 209
      - IndexOutOfRangeException 211
      - InvalidCastException 349, 391, 591
      - InvalidOperationException 597, 606
      - KeyNotFoundException 602
      - Message property **211**
      - parameter 211
    - Exception Assistant **370**
    - Exception class **372**
    - exception filter (C# 6) **392**
      - when clause 392
    - exception handler **363**, 373
    - exception handling 58
    - .exe file name extension **51**
    - executable 8, **51**
    - execute an application 47
    - ExecutionEngineException class **372**
    - exhausting memory 189
    - Exists method of class
      - Directory 558
    - Exists method of class
      - DirectoryInfo **479**
    - Exit method of class
      - Application **448**, 461
    - exit point of a control statement 99
    - Exp method of Math 154
    - expand a tree 25
    - Expand method of class TreeNode 470
    - expand node 469
    - ExpandAll method of class
      - TreeNode 470
    - explicit conversion **113**
    - explicit type argument **573**
    - exponential complexity **678**
    - exponential method 154
    - exponentiation operator 130
    - expression **59**, 99, 114
    - expression bodied (C# 6)
      - => **185**
      - method **185**
      - property **185**
    - expression lambda **613**
  - extend a class **300**
  - extensibility 330
  - extensible programming
    - language **68**
  - extension method **255**, **295**, 608, 618, 637
    - Aggregate 616, **619**
    - Enumerable class 616
    - IEnumerable<T> 616
    - LINQ 608, 616
    - LINQ to Entities 649
    - Max **619**
    - Min **619**
    - OrderBy **621**
    - Reverse **507**
    - Select 616, **622**
    - Sum **619**
    - Where 616, **621**
  - external iteration **615**
- F**
- F format specifier 92
    - for floating-point numbers **114**
  - factorial 186
  - Factorial method 187
  - false keyword 61, **99**, 100
  - fault tolerant **58**
  - fault-tolerant program **211**, **363**
  - Fibonacci series 676, 678
  - field
    - default initial value 73
    - in a database table **631**
  - field of a class **154**, 175
  - field width **131**
  - File class **557**, 561
    - Delete method **565**
  - File class methods (partial list) 557
  - File menu 21
  - File name extensions
    - .cs 25
    - .csproj 36
  - FileAccess enumeration 539
  - FileInfo class **479**
    - FullName property **479**
    - Name property **479**
  - file-position pointer 544
  - files **530**



- FileStream class **531**, 538, 544, 555
  - Seek method 549
- FillEllipse method of class Graphics **433**, **468**
- FillPie method of class Graphics **468**
- FillRectangle method of class Graphics **468**
- filter (functional programming) **616**, 621
- filter a collection using LINQ **247**
- filter elements 621
- filtering array elements 611
- final state in the UML **98**
- Finalize method
  - of class object 325
- finally block **370**, 374
- finally blocks always execute, even when no exception occurs 375
- Find method of class LinkedList **607**
- Finished design of MasterDetail app 662
- First extension method of interface IEnumerable<T> **255**
- FirstDayOfWeek property of class MonthCalendar 449
- FirstNode property of class TreeNode 469
- Fisher-Yates shuffling algorithm 215
- flag value **110**
- flash drive 530
- FlatButton property of class Button 411
- Flickr 687
- Flickr API key 687
- float simple type 57, **110**, 131, 702
- floating-point division 114
- floating-point number **110**, 112
  - double data type **110**
  - float data type **110**
- Floor method of Math 153
- flow of control 106, 112
- flow of control in the if...else statement 100
- focus **397**
- Focused property of class Control 406
- Font class **418**
  - Style property **418**
- Font dialog **34**
- Font property of a Label **33**
- Font property of class Control 407
- Font property of class Form 398
- font size 33
- font style 34, 416
- Font window 34
- FontStyle enumeration **418**
- for iteration statement 98, **125**, 126, 128, 129
  - activity diagram 128
  - header **126**
- foreach iteration statement **203**, 597
  - on rectangular arrays 234
- ForeColor property of class Control 407
- foreign key **634**, 636
- form **20**
- form background color 32
- Form class 397
  - AcceptButton property 398
  - ActiveMdiChild property 486
  - AutoScroll property 398
  - CancelButton property 398
  - Close method 398
  - Font property 398
  - FormBorderStyle property 398
  - Hide method 398
  - IsMdiChild property 486
  - IsMdiContainer property **485**, 486
  - LayoutMdi method 486, **487**
  - Load event 398
  - MaximumSize property **409**
  - MdiChildActivate event 486
  - MdiChildren property 486
  - MdiParent property **485**
  - MinimumSize property **409**
  - Padding property **408**
  - Show method 398
  - Text property 398
- Form properties, methods and events 398
- format item **522**
- Format menu 21
- Format property of class DateTimePicker **450**
- format specifier **91**
  - C for currency **91**
  - D for integer values 92, 206
  - E for scientific notation 92
  - F for floating-point numbers 92, **114**
  - G for scientific or floating-point notation depending on the context 92
  - N for numbers 92
  - table 92
- format string **522**
- FormatException class **366**, 369
- formatted output
  - field width **131**
  - left align **131**
  - right align **131**
- FormBorderStyle property of class Form 398
- forward slash character (/) in end tags **692**
- fragile software **320**
- Framework Class Library 574
- from clause of a LINQ query **249**
- FromStream method of class Image 694
- FullName property of class DirectoryInfo **479**
- FullName property of class FileInfo **479**
- FullName property of class Type 326
- FullPath property of class TreeNode 469
- FullStackException indicates a stack is full 580
- fully qualified class name 162, 399, 494
- fully qualified name **158**, 162, 399
- Func delegate **619**, 621, 649, 650, 655
- Func<TResult> delegate **679**

function key **436**  
 functional programming **6**, **247**,  
 590, **615**  
 filter **616**  
 map **616**  
 reduce **616**

## G

G format specifier **92**  
 game playing **164**  
 garbage collector **284**, **374**  
 general catch clause **370**, **372**,  
 392  
 general class average problem  
 110  
 generic class **257**, **568**, **577**  
 Dictionary **592**  
 LinkedList **592**, **603**  
 LinkedListNode **603**  
 List **592**  
 Queue **592**  
 SortedDictionary **592**, **599**,  
 601  
 SortedList **592**  
 SortedSet **627**  
 Stack **592**  
 generic interface **568**  
 generic method **568**, **571**  
 creating **584**  
 implementation **571**  
 generic programming **590**  
 generics **568**  
 class **568**  
 class constraint **576**  
 compile-time type safety **568**  
 constructor constraint  
 (new()) **576**  
 default type constraint  
 (object) of a type  
 parameter **579**  
 interface **568**  
 interface constraint **576**  
 method **571**  
 overloading **577**  
 reference type constraint  
 class **576**  
 reusability **577**  
 scope of a type parameter  
 579

generics (cont.)  
 specifying type constraints **574**  
 Stack class **577**  
 type argument **573**, **581**  
 type checking **568**  
 type constraint of a type  
 parameter **574**, **576**  
 type parameter **572**  
 type parameter list **572**  
 value type constraint struct  
**576**  
 where clause **576**  
 get accessor of a property **4**, **79**,  
 80, **81**  
 get keyword **82**  
 GetByteArrayAsync method of  
 class HttpClient **694**  
 GetCreationTime method of  
 class Directory **558**  
 GetCreationTime method of  
 class File **557**  
 GetDirectories method of class  
 Directory **473**, **558**  
 GetDirectories method of class  
 DirectoryInfo **479**  
 GetEnumerator method of  
 interface IEnumerable **596**  
 GetExtension method of class  
 Path **565**  
 GetFiles method of class  
 Directory **558**, **565**  
 GetFiles method of class  
 DirectoryInfo **479**  
 GetHashCode method of class  
 Object **599**  
 GetItemChecked method of class  
 CheckedListBox **462**  
 GetLastAccessTime method of  
 class Directory **558**  
 GetLastAccessTime method of  
 class File **557**  
 GetLastWriteTime method of  
 class Directory **558**  
 GetLastWriteTime method of  
 class File **557**  
 GetLength method of an array **229**  
 GetNodeCount method of class  
 TreeNode **470**  
 GetObject method of class  
 ResourceManager **426**

GetSelected method of class  
 ListBox **458**  
 GetStringAsync method of class  
 HttpClient **691**, **692**  
 getter-only auto-implemented  
 properties (C# **6**) **212**  
 getter-only auto-implemented  
 property **282**, **289**  
 GetType method of class object  
 326, **350**  
 GetValueOrDefault method of a  
 nullable type **391**  
 GIF (Graphic Interchange  
 Format) **36**  
 global namespace **162**  
 Good Programming Practices  
 overview xxvii  
 goto elimination **97**  
 goto statement **96**  
 graph information **206**  
 Graphic Interchange Format  
 (GIF) **36**  
 graphical user interface (GUI)  
 19, **163**, **395**  
 Graphics class **433**, **467**  
 Clear method **467**  
 DrawEllipse method **468**  
 DrawPie method **468**  
 DrawRectangle method **468**  
 FillEllipse method **433**,  
**468**  
 FillPie method **468**  
 FillRectangle method **468**  
 Graphics property of class  
 PaintEventArgs **497**, **498**  
 group by (LINQ) **565**  
 GroupBox class **12**  
 GroupBox control **413**  
 Controls property **414**  
 properties **413**  
 Text property **413**  
 guard condition in the UML **99**  
 GUI (graphical user interface)  
**19**, **395**  
 container control **407**  
 control **396**  
 control, basic examples **396**  
 thread **622**  
 Windows Forms **396**  
 guillemets (« and ») **87**

**H**

handle an event 403  
 handle an exception **367**  
 hard disk 530  
*has-a* relationship **280, 300**  
 hash function **599**  
 hash table **598**  
 hashing **598**  
 Hashtable class 593  
 HasValue property of a nullable type **391**  
 Height property of structure Size **409**  
 Help menu 22, **28**  
 HelpLink property of Exception 383  
 “hidden” fields 175  
 hide implementation details 270  
 Hide method of class Control 407  
 Hide method of class Form 398  
 HoloLens 10  
 horizontal tab 54  
 hot key 439  
 HourlyEmployee class that extends Employee 341  
 HttpClient class 691  
   CancelPendingRequests method **691**  
   GetByteArrayAsync method **694**  
   GetStringAsync method **691, 692**

**I**

IBM  
   DB2 630  
 ICollection<T> interface 591  
 IComparable interface 360  
 IComparable<T> interface **574, 627**  
   CompareTo method **574**  
 IComparer<T> interface **627**  
 IComponent interface 360, **396**  
 icon 22  
 IDE (Integrated Development Environment) 10, 16  
 identifier **44, 55**  
 identity column in a database table **633**  
 IDictionary<K,V> interface 591  
 IDisposable interface 361, 381  
   Dispose method 361  
 IEC 60559 702  
 IEEE 754 702  
 IEnumerable interface  
   method GetEnumerator **596**  
 IEnumerable<T> extension  
   method  
   Any **255**  
   Average **619, 625**  
   Count **255**  
   Distinct **255**  
   First **255**  
   Max 625  
   Min 625  
 IEnumerable<T> interface **251, 591, 596, 637**  
 IEnumerator interface 361, **596**  
 if single-selection statement 61, 64, 98, 99, 100, 133  
   activity diagram 100  
 if...else double-selection statement 98, 100, 101, 112, 133  
   activity diagram 101  
 ignoring array element zero 210  
 IList<T> interface 591  
 Image property of class PictureBox **35, 424**  
 image resource **426**  
 ImageIndex property of class ListViewItem **475**  
 ImageIndex property of class TreeNode 469  
 ImageList class **469, 475**  
   Images property **475**  
 ImageList property of class TabControl 481  
 ImageList property of class TreeView 469  
 Images Collection Editor 475  
 Images property of class ImageList **475**  
 imaginary part of a complex number 292  
 immutability **616**  
 immutable string **286, 506**  
 imperative programming **247**

implement an interface **329, 353, 358**  
 implementation-dependent code 270  
 implementing a Dispose method (link to MSDN article) 361  
 implicit conversion **114**  
 implicit conversions between simple types 161  
 implicitly typed local variable **206, 207, 250, 256**  
 in parallel **673**  
 In property of class Console 530  
 increment 129  
   a control variable **124**  
   expression 142  
 increment and decrement operators 118  
 increment operator, ++ 118  
 Increment property of class NumericUpDown **428**  
 indefinite repetition **110**  
 indentation 46, 64, 100, 102  
   indent size 45  
 independent software vendor (ISV) 324  
 index **197, 209**  
 index initializer (C# 6) **608**  
 Index property of class ItemEventArgs 462  
 index zero **197**  
 indexer 506  
   [] operator **591**  
   of a SortedDictionary<K,V> 602  
 IndexOf method  
   of class Array **597**  
   of class List<T> 257  
   of class string 511, 513  
 IndexOfAny method of class string 511  
 IndexOutOfRangeException class 209, **211, 372**  
 indirect base class 300, 302  
 infer a local variable’s type 206  
 infinite loop **106, 113, 127, 189**  
 infinite recursion **189**  
 infinity symbol 636  
 information hiding **4, 75, 351**

- inherit from class `Control` 498
- inherit from Windows Form control 498
- inheritance **4**, **300**, 305, 324
  - examples 301
  - hierarchy **301**
  - hierarchy for class `Shape` 302
  - hierarchy for university
    - `CommunityMembers` 302
  - single **300**
  - with exceptions 373
- initial state in the UML **98**
- initial value of control variable **124**
- `InitialDelay` property of class `ToolTip` 427
- initializer list **200**
- Initializing jagged and rectangular arrays 227
- Initializing the elements of an array with an array initializer 200
- initializing two-dimensional arrays in declarations 227
- inlining method calls **351**
- `InnerException` property of `Exception` **382**, 386
- innermost set of brackets 210
- input data from the keyboard 65
- input validation 369
- `Insert` method of class `List<T>` **260**
- `Insert Separator` option 442
- `Insert Snippet` window 174
- inserting separators in a menu 442
- instance **3**
- instance variable **4**, 72, 73, 154
- `int` operands promoted to `double` 114
- `int` simple type **57**, 117, 702
  - `Parse` method **58**
  - `TryParse` method **369**
- `Int32` struct 574
- integer **56**
  - division **60**, **110**
  - value **57**
- integer array 200
- integer division without exception handling 364
- integer promotion **114**
- Integrated Development Environment (IDE) **10**, 16
- `IntelliSense` **49**, 249, 297, 298, 631, 636
- interest rate 129
- interface **251**, 329, 353, 359
  - declaration **352**
- interface constraint **576**
- interface keyword **352**, 359
- Interfaces
  - `ICollection<T>` 591
  - `IComparable` 360, **574**
  - `IComparable<T>` **627**
  - `IComparer<T>` **627**
  - `IComponent` 360, **396**
  - `IDictionary` 591
  - `IDictionary<K, V>` 591
  - `IDisposable` 361, 381
  - `IEnumerable` **596**
  - `IEnumerable<T>` **251**, 591, 637
  - `IEnumerator` 361, **596**
  - `IList<T>` 591
  - `IQueryable<T>` **637**
  - `ISerializable` **550**
- interpolation
  - `$` before a string literal **56**
  - string **55**, 56
- interpolation expression **56**, 59, 64
  - calculation in 59
  - specify a format 91
- `Interval` property of class `Timer` **499**
- `InvalidCastException` class 349, 391, 591
- `InvalidOperationException` class 597, 606
- `Invoice` class implements `IPayable` 355
- `Invoke` method of class `Control` **686**
- `InvokeRequired` property of class `Control` **686**
- iOS 9
- `IPayable` interface declaration 355
- `IPayable` interface hierarchy
  - UML class diagram 354
- `IQueryable<T>` interface **637**
- is-a* relationship **300**, 330
- `is` operator **349**
- `IsDigit` method of `Char` 526
- `ISerializable` interface **550**
- `IsLetter` method of `Char` 527
- `IsLetterOrDigit` method of `Char` 527
- `IsLower` method of `Char` 527
- `IsMdiChild` property of class `Form` 486
- `IsMdiContainer` property of class `Form` **485**, 486
- `IsPunctuation` method of struct `Char` 527
- `IsSymbol` method of struct `Char` 527
- `IsUpper` method of struct `Char` 527
- `IsWhiteSpace` method of struct `Char` 527
- `ItemActivate` event of class `ListView` 474
- `ItemCheck` event of class `CheckedListBox` **461**, 462
- `ItemCheckEventArgs` class 462
  - `CurrentValue` property 462
  - `Index` property 462
  - `NewValue` property 462
- `Items` property
  - of class `ComboBox` **465**
  - of class `ListBox` 457, **458**
  - of class `ListView` 474
- `ItemSize` property of class `TabControl` 481
- iteration **109**
  - counter controlled 115
  - of a loop 142
  - sentinel controlled 110
- iteration (looping)
  - of a for loop 210
- iteration statement **97**, **98**, 106
  - `do...while` 98, 132, 133, 133
  - for 98, 128
  - foreach 98
  - while 98, 107
- iteration terminates 106
- iteration variable **203**
- iterative 189

**J**

Jacopini, G. 97  
 jagged array **225**, 226, 227  
 JIT (just-in-time) compilation **8**  
 joining database tables **635**, 658  
 LINQ to Entities 655  
 Joint Photographic Experts  
 Group (JPEG) 36  
 JPEG (Joint Photographic  
 Experts Group) 36  
 just-in-time (JIT) compiler **8**

**K**

key code **436**  
 key data 436  
 key event **433**, 434  
 key value 436  
 keyboard 56, 396  
 keyboard shortcuts **439**  
 KeyChar property of class  
 KeyEventArgs **433**  
 KeyCode property of class  
 KeyEventArgs 434  
 KeyData property of class  
 KeyEventArgs 434  
 KeyDown event of class Control  
**433**, 434  
 KeyEventArgs class **433**  
 Alt property 434, 436  
 Control property 434, 436  
 KeyCode property 434  
 KeyData property 434  
 KeyValue property 434  
 Modifiers property 434  
 properties 434  
 Shift property 434, 436  
 KeyNotFoundException class **602**  
 KeyPress event of class Control  
**433**, 434  
 KeyEventArgs class **433**  
 KeyChar property **433**, 434  
 properties 434  
 keys  
 function **436**  
 modifier **433**  
 Keys enumeration **433**  
 Keys property  
 of Dictionary **565**  
 of SortedDictionary<K, V>  
**602**

KeyUp event of class Control  
**433**, 434  
 KeyValue property of class  
 KeyEventArgs 434  
 KeyValuePair<K, V> structure **602**  
 Keywords **44**, 98  
 abstract 316, **334**  
 as 349, 391  
 async 674, **675**, 678  
 await **675**, 679  
 base **303**, **316**, 323, 324  
 break **136**  
 case **136**  
 char **57**  
 class **44**  
 const 139, **154**, 202, 288  
 continue **141**  
 decimal **57**, **87**  
 default **136**  
 delegate **403**, 610  
 do **132**  
 else 100  
 enum **172**  
 float **57**  
 for **125**  
 get **82**  
 if **61**, 99, 100  
 int **57**  
 interface **352**  
 is **349**  
 nameof **276**  
 namespace 493  
 new **70**, 198, 226  
 null **73**, 121, 189, 198  
 operator **293**  
 out **191**  
 override **213**, **308**, 316  
 params **236**  
 partial 401  
 private **75**, 270, 303  
 protected 75, 270, **303**  
 public **75**, **267**, 270, 303  
 readonly **288**  
 ref **191**, 217  
 return **75**, 160  
 sealed **351**  
 set **82**  
 static **130**, 157  
 struct **292**  
 this **271**, 285

Keywords (cont.)  
 value (contextual) **82**  
 var **206**  
 virtual **316**  
 void **46**, **73**  
 while 106, 132

**L**

Label 33  
 label 410  
 Label class **20**, **29**, 32  
 Label control 396, 410  
 label in a switch **136**, 137  
 lambda expression **611**, 621,  
 650, 655, 675, 679  
 expression lambda **613**  
 lambda operator (=>) **613**,  
 650  
 statement lambda **614**  
 lambda operator **613**  
 language independence **8**  
 Language Integrated Query  
 (LINQ) **246**  
 language interoperability **8**  
 LargeImageList property of  
 class ListView **475**  
 last-in first-out (LIFO) order 584  
 Last property of class  
 LinkedList **607**  
 last-in, first-out (LIFO) **177**  
 LastIndexOf method of class  
 Array **597**  
 LastIndexOf method of class  
 string 511, 513  
 LastIndexOfAny method of class  
 string 511  
 LastNode property of class  
 TreeNode 469  
 late binding **348**  
 layout, control 406  
 LayoutMdi method of class Form  
 486, **487**  
 leading 0 **206**  
 left align output **131**  
 left brace, { **45**, 46, 57  
 legacy code **590**  
 Length property  
 of an array **198**, 198  
 of class string 506, 507  
 of class StringBuilder 518

- let clause of a LINQ query **263**
- LIFO (last-in, first-out) 177, 584
- line numbers xxxv
- LinkArea property of class
  - LinkLabel 453
- LinkBehavior property of class
  - LinkLabel 454
- LinkClicked event of class
  - LinkLabel 453, 454
- LinkColor property of class
  - LinkLabel 454
- LinkedList generic class 592
  - AddFirst method **606**
  - AddLast method **605**
  - method Find **607**
  - method Remove **607**
  - property First 606
  - property Last **607**
- LinkedList<T> class 592
- LinkedList<T> generic class **603**
- LinkedListNode class
  - property Next **603**
  - property Previous **603**
  - property Value **603**
- LinkedListNode<T> generic class **603**
- LinkLabel class 439, **453**
  - ActiveLinkColor property 453
  - LinkArea property 453
  - LinkBehavior property 454
  - LinkClicked event 453, 454
  - LinkColor property 454
  - LinkVisited property 454
  - Text property 454
  - UseMnemonic property 454
  - VisitedLinkColor property 454
- LinkLabel properties and an event 453
- LinkVisited property of class
  - LinkLabel 454
- LINQ (Language Integrated Query) **246**, 561, 615
  - anonymous type **256**
  - ascending modifier **250**
  - Count method 565
  - deferred execution **263**
  - descending modifier **250**
- LINQ (Language Integrated Query) (cont.)
  - extension method 616
  - extension method Aggregate **619**
  - extension method Max **619**
  - extension method Min **619**
  - extension method Select **622**
  - extension method Sum **619**
  - extension method Where **621**
  - extension methods for arithmetic 619
  - from clause **249**
  - group by **565**
  - let clause **263**
  - LINQ to Entities 247, **631**
  - LINQ to Objects **246**, 561, 631
  - LINQ to XML 247
  - orderby clause **250**
  - provider **247**
  - query expression **246**
  - query syntax 616, 637
  - range variable **249**
  - Resource Center 671
  - select clause **250**
  - usage throughout the book 247
  - where clause **250**
- LINQ to Entities 247, **631**, 632, 649
  - data binding **637**
  - DbContext class **637**, **643**
  - extension methods 649
  - Object data source **645**
  - primary keys 631
  - saving changes back to a database 650
- LINQ to Objects **246**, 247, 588, 590, 623, 631, 637
  - using a List<T> 261
  - using an array of Employee objects 252
  - using an int array 248
- LINQ to XML 247, 687, 691
- LINQPad ([www.linqpad.net](http://www.linqpad.net)) 671
- Linux 9
- List class 592
- list, editable 465
- List<T> generic class **256**, 592
  - Add method **260**
  - AddRange method 257
  - Capacity property **257**
  - Clear method 257
  - Contains method 257, **260**
  - Count property **257**
  - IndexOf method 257
  - Insert method **260**
  - methods 257
  - Remove method 257, **260**
  - RemoveAt method 257, **260**
  - RemoveRange method 257
  - Sort method 257
  - TrimExcess method 257
- ListBox control 396, 439, **456**
  - ClearSelected method 458
  - GetSelected method **458**
  - Items property **458**
  - MultiColumn property 457
  - properties, method and event 457
  - SelectedIndex property **458**
  - SelectedIndexChanged event **457**
  - SelectedIndices property **458**
  - SelectedItem property **458**
  - SelectedItems property **458**
  - SelectionMode property **457**
  - Sorted property 458
- ListBox.ObjectCollection class 458
- ListView control **474**
  - Activation property **474**
  - CheckBoxes property **474**
  - ItemActivate event 474
  - Items property 474
  - LargeImageList property **475**
  - MultiSelect property **474**
  - SelectedItems property 474
  - SmallImageList property **475**
  - View property **474**
- ListView displaying files and folders 475
- ListView properties and events 474

- ListViewItem class **475**
    - ImageIndex property **475**
  - literal 46
  - Load event of class Form 398
  - Load extension method of class
    - DbExtensions **650**, 654
  - load factor **599**
  - local variable **74**, 108, 109, 174, 175, 176, 272
  - local variable “goes out of scope” 530
  - Location property of class
    - Control 409
  - Log method of Math 154
  - logarithm 154
  - logic error 58
  - logical negation, ! 146
    - operator truth table 146
  - logical operators 143, 146, 147
  - logical output operation **531**
  - logical XOR, ^ **145**
  - long simple type 702
  - long-term retention of data 530
  - Long value of enumeration
    - DateTimePickerFormat **450**
  - loop 107
    - body 132
    - continuation condition **98**
    - counter 124
    - infinite **106**, 113
  - loop-continuation condition **124**, 126, 132, 133, 142
  - looping **109**
  - lowercase letter 44
- M**
- m*-by-*n* array **225**
  - m* to indicate a decimal literal **89**
  - magic numbers **202**
  - Main method 46, 57
  - make your point (game of craps) 169
  - managed code 7
  - many-to-many relationship **636**, 643
  - map (functional programming) **616**, 622
  - map elements to new values 622
  - MariaDB 630
  - mask the user input 410
  - master/detail view **661**
  - Match class 503, 527
  - Math class 153
    - Abs method 153
    - Ceiling method 153
    - Cos method 153
    - E constant **154**
    - Exp method 154
    - Floor method 153
    - Log method 154
    - Max method 154
    - Min method 154
    - PI constant **154**
    - Pow method 130, 131, 153, 154
    - Sin method 153
    - Sqrt method 153, 154, 160, 388
    - Tan method 153
  - Max IEnumerable<T> extension method **619**, 625
  - Max method of Math 154
  - Max ParallelQuery<T> extension method 625
  - MaxDate property of class
    - DateTimePicker 450, **453**
  - MaxDate property of class
    - MonthCalendar 449
  - MaxDropDownItems property of class
    - ComboBox **464**
  - Maximum property of class
    - NumericUpDown **428**
  - MaximumSize property of class
    - Control 409
  - MaximumSize property of class
    - Form **409**
  - MaxSelectionCount property of class
    - MonthCalendar 449
  - .mdf file extension 632
  - MDI (Multiple Document Interface) **484**
    - child 491
    - parent and child properties, method and event 486
    - title bar 487
    - window **396**
  - MdiChildActivate event of class
    - Form 486
  - MdiChildren property of class
    - Form 485, 486
  - MdiLayout enumeration **487**
    - ArrangeIcons value 487
    - Cascade value 487
    - TileHorizontal value 487
    - TileVertical value 487
  - MdiParent property of class Form **485**
  - MdiWindowListItem property of class
    - MenuStrip **487**
  - member access (.) operator 130, 153, 285
  - member access operator (.) **70**
  - MemberwiseClone method of class object 326
  - memory consumption 589
  - memory leak **284**, **374**
  - MemoryStream class **531**
  - menu 21, 396, 439
    - access shortcut 439
    - access shortcut, create 440
    - Analyze 21
    - Build 21
    - Debug 21
    - Edit 21
    - ellipsis convention 442
    - expanded and checked 440
    - File 21
    - Format 21
    - Help 22, **28**
    - Project 21
    - Team 21
    - Test 21
    - Tools 21
    - View 21, 24
    - Window 22
  - menu bar **21**, 396
    - in Visual Studio IDE 21
  - menu item 21, 439
  - MenuItem property
    - MdiWindowListItem example 487
  - MenuStrip class **440**
    - MdiWindowListItem property **487**
    - RightToLeft property 443
  - MenuStrip properties and events 443
  - merge symbol in the UML **106**
  - message **46**

- Message property of Exception **211**, 379, 382
  - method **3**, **46**
    - local variable **74**
    - parameter **74**
    - parameter list **74**
    - static **130**
  - method call **3**, 157
  - method-call stack **177**, 382
  - method declaration 157
  - method header 73
  - method overloading **181**
  - method parameter list 236
  - MethodInvoker delegate 686
  - methods implicitly sealed 351
  - methods of class List<T> 257
  - Metro 9
  - Microsoft
    - SQL Server 630
  - Microsoft Developer Network (MSDN) **18**
  - Microsoft Intermediate Language (MSIL) **8**, 51
  - Microsoft Visual Studio
    - Community edition xxxiii
  - Min IEnumerable<T> extension
    - method **619**, 625
  - Min LINQ extension method
  - Min method of Math 154
  - Min ParallelQuery<T> extension
    - method 625
  - MinDate property of class
    - DateTimePicker 450, **453**
  - MinDate property of class
    - MonthCalendar 449
  - minimized and maximized child window 486
  - Minimum property of class
    - NumericUpDown **428**
  - MinimumSize property of class
    - Control 409
  - MinimumSize property of class
    - Form **409**
  - mobile application 2
  - modal dialog **538**
  - model 639
  - model designer 642
  - modifier key **433**
  - Modifiers property of class
    - EventArgs 434
  - modulus operator (%) **59**
  - monetary amount 87
  - monetary calculations 131
  - monetizing your apps 10
  - Mono Project **xxxv**, **6**
  - MonthCalendar class **449**
    - DateChanged event **449**
    - FirstDayOfWeek property 449
    - MaxDate property 449
    - MaxSelectionCount property 449
    - MinDate property 449
    - MonthlyBo1dedDates property 449
    - SelectionEnd property 449
    - SelectionRange property 449
    - SelectionStart property 449
  - MonthlyBo1dedDates property of class MonthCalendar 449
  - More Windows...** option in Visual Studio .NET 487
  - mouse 396
    - pointer 23
  - mouse click **430**
  - mouse event **430**, 431
  - mouse move **430**
  - MouseDown event of class Control 431
  - MouseEnter event of class Control 431
  - MouseEventArgs class **430**
    - Button property 431
    - Clicks property 431
    - X property 431
    - Y property 431
  - MouseEventArgs properties 431
  - MouseEventHandler delegate **430**
  - MouseHover event of class Control 431
  - MouseLeave event of class Control 431
  - MouseMove event of class Control 431
  - MouseUp event of class Control 431
- MouseWheel event of class Control 431
  - Move method of class Directory 558
  - Move method of class File 557
  - MoveFirst method of class BindingSource **654**
  - MoveNext method of IEnumerator **596**
  - MSDN (Microsoft Developers Network) **18**
  - MSIL (Microsoft Intermediate Language) **8**
  - multicast delegate **404**
  - multicast event 404
  - MultiCastDelegate class **404**
  - MultiColumn property of class ListBox 457
  - multidimensional array 225
  - MultiExtended value of enumeration SelectionMode **457**
  - MultiLine property of class TabControl 481
  - MultiLine property of class TextBox 411
  - multiple document interface (MDI) **396**, **484**
  - multiple-selection statement 98
  - multiplication, \* **59**
  - MultiSelect property of class ListView **474**
  - MultiSimple value of enumeration SelectionMode **457**
  - multithreading **674**
  - mutual exclusion **419**
  - mutually exclusive options 419
  - MySQL 630
- ## N
- N format specifier 92
  - name collision **399**, 494
  - name conflict **399**, 494
  - Name property of class DirectoryInfo **479**
  - Name property of class FileInfo **479**
  - named constant **202**
  - named parameter **185**



- nameof operator (C# 6) **276**
  - namespace **43**, 162, 493
    - declaration 493
    - keyword 493
  - namespace declaration 399
  - Namespaces
    - of the FCL 162
    - System 164
    - System.Collections 163, 574, **590**
    - System.Collections.Concurrent **590**
    - System.Collections.Generic 163, 256, 565, **590**
    - System.Collections.Specialized **590**
    - System.Data.Entity 163, **637**
    - System.Data.Linq 163
    - System.Diagnostics **456**
    - System.Drawing 418
    - System.IO 163, 531
    - System.Linq 163, 248, 637
    - System.Net.Http 691
    - System.Runtime.Serialization.Formatters.Binary **550**
    - System.Runtime.Serialization.Json 550
    - System.Text 163, 503
    - System.Text.RegularExpressions 503
    - System.Threading.Tasks **679**
    - System.Web 163
    - System.Windows.Controls 163
    - System.Windows.Forms 163, 397
    - System.Windows.Input 163
    - System.Windows.Media 163
    - System.Windows.Shapes 163
    - System.Xml 163
    - System.Xml.Linq 163, **693**
    - System.Xml.Serialization 550
  - naming convention for methods that return boolean 139
  - NaN constant of structure Double **366**, 388
  - natural logarithm 154
  - navigation property **637**, 646, 647
  - NegativeInfinity constant of structure Double **366**
  - NegativeNumberException
    - represents exceptions caused by illegal operations performed on negative numbers 387
  - nested array initializer 225
  - nested control statements 114, 168
  - nested for statement 206, 227, 229
  - nested foreach statement 229
  - nested if selection statement 104, 105
  - nested if...else selection statement 101, 104, 105
  - nested parentheses 60
  - new keyword **70**, 198, 226
  - New Project** dialog **19**, 20, 29
  - new() (constructor constraint) **576**
  - newline character **54**
  - newline escape sequence, \n 54, 57
  - newValue property of class ItemCheckEventArgs 462
  - Next method of class Random **164**, 165, 168
  - Next property of class LinkedListNode **603**
  - NextNode property of class TreeNode 469
  - node **468**
    - child **468**
    - expand and collapse 469
    - parent **468**
    - root **468**
    - sibling **468**
  - Nodes property
    - of class TreeNode 469
    - of class TreeView **469**
  - non-static class member 285
  - None value of enumeration SelectionMode **457**
  - not selected state **419**
  - note (in the UML) 98
  - Notepad 454
  - Now property of DateTime **625**
  - Now property of structure DateTime **499**
  - NuGet package manager **644**
  - null coalescing operator (??) 391, **392**
  - null keyword **73**, 121, 189, 198
  - nullable type **391**, 607
    - GetValueOrDefault method **391**
    - HasValue property **391**
    - Value property **391**
  - null-conditional operator (?.) **390**, 391, 607
  - null-conditional operator (?[]) **607**
  - NullReferenceException class **372**
  - numbers with decimal points 87
  - numeric literal
    - whole number 130
    - with a decimal point 130
  - NumericUpDown control 396, **428**
    - DecimalPlaces property **428**
    - Increment property **428**
    - Maximum property **428**
    - Minimum property **428**
    - ReadOnly property **430**
    - UpDownAlign property 429
    - Value property 429
    - ValueChanged event 429
  - NumericUpDown properties and events 428
- ## O
- object **2**
  - Object Browser (Visual Studio .NET) 289
  - object class **300**, **305**, 325
    - Equals method 325
    - Finalize method 325
    - GetHashCode method 326
    - GetType method 326, **350**
    - MemberwiseClone method 326
    - ReferenceEquals method 326
    - ToString method 307, 326

- object-creation expression **70**, 85
- Object data source **645**
- object initializer **291**
- object initializer list **291**
- object methods that are
  - inherited directly or indirectly by all classes **325**
- object of a class **4**
- object of a derived class **331**
- object of a derived class is instantiated **324**
- object-oriented analysis and design (OOAD) **5**
- object-oriented language **5**
- object-oriented programming (OOP) **5**, **300**
- object serialization **550**
- ObjectCollection collection
  - Add method **459**
  - Clear method **461**
  - RemoveAt method **459**
- object-oriented programming **590**
- ObservableCollection<T> class **650**, **655**
- one-to-many relationship **635**, **636**
- One value of enumeration
  - SelectionMode **457**
- OnPaint method of class Control **497**
- OOAD (object-oriented analysis and design) **5**
- OOP (object-oriented programming) **5**, **300**
- Open method of class File **557**
- OpenFileDialog class **543**, **549**
- opening a project **21**
- OpenRead method of class File **557**
- OpenText method of class File **557**
- OpenWrite method of class File **557**
- operand **58**
- operands of a binary operator **59**
- operating system **8**
- operation in the UML **76**
- operation parameter in the UML **77**
- operator **59**
- operator keyword **293**
- operator overloading **291**
- operator precedence **60**
  - operator precedence chart **114**
  - rules **60**
- Operators **613**, **650**
  - ^, boolean logical exclusive OR **143**, **145**
  - , prefix decrement/postfix decrement **118**, **119**
  - , subtraction **60**, **61**
  - !, logical negation **143**, **146**
  - !=, not equals **62**
  - ?:, ternary conditional operator **103**, **120**
  - \*, multiplication **60**, **61**
  - \*=, multiplication compound assignment **118**
  - /, division **60**, **61**
  - \=, division compound assignment **118**
  - &, boolean logical AND **143**, **145**
  - &&, conditional AND **143**, **144**
  - %, remainder **60**, **61**
  - %=, remainder compound assignment **118**
  - +, addition **60**, **61**
  - ++, prefix increment/postfix increment **118**, **119**
  - +=, addition assignment operator **117**
  - +=, addition compound assignment **117**
  - <, less than **62**
  - <=, less than or equal **62**
  - =, assignment operator **58**
  - =, subtraction compound assignment **117**
  - ==, is equal to **62**
  - >, greater than **62**
  - >=, greater than or equal to **62**
- Operators (cont.)
  - |, boolean logical inclusive OR **143**, **145**
  - ||, conditional OR **143**, **144**
  - arithmetic **59**
  - as **349**, **391**
  - await **675**
  - binary **58**, **59**
  - boolean logical AND, & **143**, **145**
  - boolean logical exclusive OR, ^ **143**, **145**
  - boolean logical inclusive OR, | **145**
  - cast **113**, **173**
  - compound assignment **117**, **120**
  - conditional AND, && **143**, **143**, **145**, **255**
  - conditional operator, ?: **103**, **120**
  - conditional OR, || **143**, **144**, **145**
  - decrement operator, -- **118**, **119**
  - increment and decrement **118**
  - increment operator, ++ **118**
  - is **349**
  - logical negation, ! **146**
  - logical operators **143**, **146**
  - logical XOR, ^ **145**
  - member access (.) **130**, **285**
  - postfix decrement **118**
  - postfix increment **118**
  - precedence chart **700**
  - prefix decrement **118**
  - prefix increment **118**
  - remainder, % **59**
- optimizing compiler **131**
- optional parameter **183**, **184**
  - default value **183**, **183**
- Oracle Corporation **630**
- orderby clause of a LINQ query **250**
  - ascending modifier **250**
  - descending modifier **250**
- OrderBy extension method **621**
- OrderBy extension method of class Queryable **649**, **655**

- out keyword **191**
  - out-of-range array index 372
  - Out property of class Console 530
  - outer set of brackets 210
  - OutOfMemoryException class **372**
  - output 54
  - output parameter **191**
  - overloaded constructors **273**
  - overloaded generic methods 577
  - overloaded methods 49, 181, 569
  - overloaded operators for
    - complex numbers 294
  - overloading constructors 86
  - override a base class method
    - 303**, 307
  - override keyword 213, **308**, 316
- P**
- package manager
    - NuGet **644**
  - Padding property of class
    - Control 409
  - Padding property of class Form
    - 408**
  - page layout software 503
  - PaintEventArgs class **497**
    - ClipRectangle property **497**
    - Graphics property **497**
  - PaintEventArgs properties 498
  - pair of braces {} 64
  - palette **32**
  - Panel class 12, 396, **413**
    - AutoScroll property 413
    - BorderStyle property 413
    - Controls property 413
    - properties 413
    - scrollbars 414
  - parallel **673**
    - operations 673
  - Parallel LINQ 247
  - ParallelEnumerable
    - AsParallel method **625**
  - ParallelEnumerable class **625**
  - ParallelQuery<T> class **625**
  - ParallelQuery<T> extension
    - method
      - Average 625
      - Max 625
      - Min 625
  - parameter **74**, 74, 85
    - output **191**
  - parameter in the UML 77
  - Parameter Info window **49**
  - parameter list **74**
    - empty 75
  - parameter name 74
  - parameter type 74
  - parameterless constructor **275**, 280, **576**
    - struct 293
  - params keyword **236**
  - parent container **409**
  - parent menu 439
  - parent node **468**
  - Parent property of class
    - DirectoryInfo **479**
  - parent window **484**
  - parentheses **46**, 60
    - nested **60**
  - Parse method
    - of simple type decimal **92**
    - of simple type double **157**
  - Parse method of class XmlDocument
    - 693**
  - Parse method of type decimal
    - 93
  - Parse method of type int **58**
  - partial class **397**
  - partial keyword 401
  - Pascal Case **44**
    - constants 202
  - Pascal case 72
  - pass an array element to a
    - method 217
  - pass an array to a method 217
  - pass-by-reference **190**
  - pass-by-value **190**, 217
  - Passing an array reference by
    - value and by reference 241
  - Passing arrays and individual
    - array elements to methods 217
  - passing options to a program
    - with command-line arguments 238
  - password TextBox **410**
  - Path class **473**, **565**
    - GetExtension method **565**
  - perform a calculation 66
  - perform a task 74, 85
  - perform an action **46**
  - performance 674
  - performing operations
    - concurrently **673**
  - permission setting 475
  - persistent
    - data **530**
  - physical output operation **531**
  - PictureBox class **20**, **29**, 35, 424, 487
    - Click event 424
    - Image property 424
    - properties and event 424
    - SizeMode property 424
  - pin icon 24
  - platform **10**
  - platform independence **8**
  - PLINQ (Parallel LINQ) 590, **623**
  - PNG (Portable Network Graphics) **36**
  - Poll analysis application 209
  - polymorphically process
    - Invoices and Employees 358
  - polymorphism 139, 326, **328**
  - pop off a stack **177**
  - portability 8
  - Portable Network Graphics (PNG) 36
  - porting **8**
  - position number 197
  - Position property of class
    - BindingSource **654**
  - PositiveInfinity constant of
    - structure Double **366**
  - postdecrement **118**
  - postfix decrement operator **118**
  - postfix increment operator **118**, 127
  - PostgreSQL 630
  - postincrement **118**
  - Pow method of Math 130, 131, 153, 154
  - power (exponent) 154
  - power of 3 larger than 100 106
  - precedence 65, 120
    - arithmetic operators 61
    - chart 61
  - precedence chart 114

- precedence chart appendix 700
  - precision
    - of double values 703
    - of float values 703
  - precision of a floating-point value **111**
  - predecrement **118**
  - predicate **250**, 621
  - prefix decrement operator **118**
  - prefix increment operator **118**
  - preincrement **118**
  - prepackaged data structures 589
  - Previous property of class
    - LinkedListNode **603**
  - PrevNode property of class
    - TreeNode 470
  - primary key **631**, 636
    - in LINQ to Entities 631
  - primitive data type promotion 114
  - principal in an interest calculation 129
  - principle of least privilege **288**
  - private
    - access modifier **75**, 271, 303
    - static class member 285
  - probability 164
  - procedural programming 590
  - Process class **456**
    - Start method **456**
  - program execution stack **177**
  - program in the general **328**
  - program in the specific **328**
  - programming paradigms
    - functional 590
    - generic 590
    - object oriented 590
    - procedural 590
    - structured 590
  - ProgressBar class **694**
  - project **18**
  - Project menu 21
  - projection **256**
  - promotion **114**, 130
    - rules **160**
  - Properties window **26**, 28, 30, 34
  - property **4**, **79**
  - property declaration **82**
  - property of a form or control **26**
  - proprietary class 324
  - protected access modifier **75**, 270, 303
  - pseudocode 100
  - pseudorandom number **165**, 168
  - public
    - access modifier **75**, 267, 303
    - interface **267**
    - member of a derived class 303
    - method 268, 270
    - service **267**
    - static class members 285
    - static method 285
  - push onto a stack **177**
- Q**
- query **246**, **630**, 632
  - query expression (LINQ) **246**
  - Queryable class **637**
    - OrderBy extension method **649**, 655
    - ThenBy extension method **650**
    - Where extension method **655**
  - Queue class 592, 593
  - Queue generic class 592
  - Queue<T> class **592**
- R**
- radians 153
  - radio button **410**, 419
    - group **419**
    - using with TabPage 484
  - RadioButton control 12, **416**, 419
    - Checked property 419
    - CheckedChanged event 419
    - properties and events 419
    - Text property 419
  - Random class 164
    - Next method **164**, 165, 168
  - random number generation 212
  - random numbers 168
    - in a range 168
    - scaling factor **165**, 168
    - seed value **165**, **169**
    - shifting value 168
  - Range method of class
    - Enumerable **625**
  - range variable of a LINQ query **249**
  - Read method of class Console 531
  - read-only property 104
  - ReadLine method of class
    - Console **58**, 70, 136, 531
  - readonly
    - keyword **288**
  - ReadOnly property of class
    - NumericUpDown **430**
  - ReadOnly property of class
    - TextBox 411
  - ReadToEnd method of class
    - StreamReader **561**
  - real number 110
  - real part of a complex number 292
  - realization in the UML **354**
  - reclaim memory 287
  - record 534, 631
  - rectangular array **225**, 226
    - with three rows and four columns 225, 226
  - recursion 473
  - recursion step 186
  - recursive call 186
  - recursive evaluation 187
    - of 5! 187
  - recursive factorial 186
  - recursive method 186
  - reduce (functional programming) **616**
  - ref keyword **191**, 217
  - refer to an object **189**
  - reference **189**
  - reference type **189**
  - reference type constraint class **576**
  - Reference, output and value parameters 192
  - ReferenceEquals method of
    - object 326
  - Regex class 503
  - regular expression **527**
  - reinventing the wheel 43
  - relational database 630, 631
  - relational database table 631
  - relational operators **61**
  - release resource 375

- release unmanaged resources 361
- remainder 60
- remainder operator, % 59, 60
- Remove method of class
  - Dictionary **565**
- Remove method of class
  - LinkedList **607**
- Remove method of class List<T> 257, **260**
- Remove method of class
  - StringBuilder 522
- RemoveAt method of class
  - List<T> 257, **260**
- RemoveAt method of class
  - ObjectCollection **459**
- RemoveRange method of class
  - List<T> 257
- Repeat method of class
  - Enumerable **697**
- repetition
  - counter controlled 112
  - sentinel controlled 112
- repetition statement
  - while 109, 112
- Replace method of class string 515, 516
- Replace method of class
  - StringBuilder 524
- representational error in floating point **131**
- requirements **5**
- requirements of an app 139
- reserved word **44**, 98
  - false **99**
  - true **99**
- Reset method of interface
  - IEnumerator **596**
- ReshowDelay property of class
  - ToolTip 427
- Resize method of class Array **198**, 246, 256
- resource **426**
- resource leak **284**, **374**
- ResourceManager class **426**
  - GetObject method **426**
- Resources class **426**
- responses to a survey 209, 210
- REST web service 687
- result of an uncaught exception **370**
- Result property of class Task **679**
- resumption model of exception handling **371**
- rethrow an exception **380**
- return keyword **75**, 160
- return statement 82, 186
- return type 75
  - of a method 73
- reusability 577
- reusable component 301
- reusable software components 2, 162
- reuse 43
- Reverse extension method **507**
- Reverse method of class Array **597**
- right align output **131**
- right brace, } 46, 57, 109, 112
- RightToLeft property of class
  - MenuStrip 443
- robust 58
- robust application **363**
- Roll a six-sided die 6,000,000 times 166
- Roll a six-sided die 60,000,000 times 208
- rolling two dice 172
- root node **468**
  - create 470
- rounding a number 60, 110, **114**, 153
  - for display 114
- row in a database table **631**
- row objects representing rows in a database table **636**
- rows of a two-dimensional array **225**
- rules of operator precedence 60
- Run command in Windows **456**
- Run method of class Task **679**, 685
- run mode **38**
- run-time logic error 58
- running an app 456
- runtime class **350**
- runtime system 577
- S**
- SalariedEmployee class that extends Employee 340
- SaveChanges method of a LINQ to Entities DbContext **637**, 651
- SaveFileDialog class 538
- saving changes back to a database in LINQ to Entities 650
- savings account 129
- sbyte simple type 702
- scaling factor (random numbers) 165, 168
- schema (database) **632**
- scope **127**, 175
  - of a declaration **174**
  - of a type parameter 579
  - static variable 285
- screen cursor **46**, 53, 54
- screen-manager program 330
- scrollbar **27**
- ScrollBars property of class
  - TextBox 411
- scrollbox **27**
- SDI (Single Document Interface) **484**
- sealed
  - class 351
  - keyword **351**
  - method 351
- secondary storage device **530**
- secondary storage devices
  - DVD 530
  - flash drive 530
  - hard disk 530
  - tape 530
- seed value (random numbers) 165, 169
- Seek method of class FileStream 549
- SeekOrigin enumeration 549
- select clause of a LINQ query **250**
- Select LINQ extension method **622**
- Select method of class Control **407**
- Select Resource dialog **35**
- selected state **419**

- SelectedImageIndex property of class `TreeNode` 470
- SelectedIndex property of class `ComboBox` **465**
- SelectedIndex property of class `ListBox` **458**
- SelectedIndex property of class `TabControl` 481
- SelectedIndexChanged event handler
  - `ComboBox` class 654
- SelectedIndexChanged event of class `ComboBox` **465**
- SelectedIndexChanged event of class `ListBox` **457**
- SelectedIndexChanged event of class `TabControl` 481
- SelectedIndices property of class `ListBox` **458**
- SelectedItem property of class `ComboBox` **465**
- SelectedItem property of class `ListBox` **458**
- SelectedItems property of class `ListBox` **458**
- SelectedItems property of class `ListView` 474
- SelectedNode property of class `TreeView` 469
- SelectedTab property of class `TabControl` 481
- selecting an item from a menu 398
- selecting data from a table 632
- selection 99
- selection statement **97, 98**
  - `if` 98, 99, 100, 133
  - `if...else` 98, 100, 101, 112, 133
  - `switch` 98, 133, 138
- SelectionEnd property of class `MonthCalendar` 449
- SelectionMode enumeration **457**
  - `MultiExtended` value 457
  - `MultiSimple` value 457
  - `None` value 457
  - `One` value 457
- SelectionMode property of class `CheckedListBox` 462
- SelectionMode property of class `ListBox` **457, 458**
- SelectionMode property of class `MonthCalendar` 449
- SelectionStart property of class `MonthCalendar` 449
- semicolon (;) **46, 56**
- sentinel-controlled iteration 110
- sentinel-controlled repetition **110, 112**
- sentinel value **110, 112**
- sentinel-controlled loop 606
- separator bar **442**
- sequence **97**
- sequence structure 97
- sequence-structure activity diagram 97
- sequential-access file 534
- sequential execution **96**
- [Serializable] attribute **550**
- SerializationException class 554
- Serialize method of class `BinaryFormatter` **550, 554**
- serialized object **550**
- service of a class 270
- set accessor of a property **4, 79, 80, 81**
- Set as Startup Project 644
- set keyword **82**
- shadow 273
- shallow copy **326**
- Shape class hierarchy 302
- shift **165**
- Shift* key 433
- Shift property of class `EventArgs` 434, 436
- Shifted and scaled random integers 166
- shifting value (random numbers) **165, 168**
- short-circuit evaluation **145**
- short simple type 702
- Short value of enumeration `DateTimePickerFormat` **450**
- shortcut key 439
- ShortcutKeyDisplayString property of class `ToolStripMenuItem` **440, 443**
- ShortcutKeys property of class `ToolStripMenuItem` **440, 443**
- shortcuts with the & symbol 442
- Show All Files icon **25**
- Show method of class `Control` 407
- Show method of class `Form` 398, 485, 491
- ShowCheckBox property of class `DateTimePicker` 450
- ShowDialog method of class `OpenFileDialog` 543, **549**
- ShowDialog method of class `SaveFileDialog` 538
- ShowShortcutKeys property of class `ToolStripMenuItem` **440, 443**
- ShowUpDown property of class `DateTimePicker` 451
- shuffling 212
  - Fisher-Yates 215
- sibling node **468**
- side effect **145, 190, 619**
- Sieve of Eratosthenes **696**
- signal value **110**
- signature of a method **182**
- simple condition **143**
- simple name 493
- simple type **57, 96, 121, 161**
  - `bool` 702
  - `byte` 702
  - `char` 57, 702
  - `decimal` **57, 87, 703**
  - `double` **57, 703**
  - `float` **57, 702**
  - `int` 57, 117, 702
  - keywords 57
  - `long` 702
  - `sbyte` 702
  - `short` 702
  - table of 702
  - `uint` 702
  - `ulong` 702
  - `ushort` 702
- Simple value of enumeration `ComboBoxStyle` 464
- Sin method of `Math` 153
- size 153
- Single Document Interface (SDI) **484**

- single-entry/single-exit control statements **99**
- single inheritance 300
- single-selection statement 98, 99
- single-line comment **42**
- single-selection statement
  - if **99**
- Size property of class Control 409
- Size structure **409**
  - Height property **409**
  - Width property **409**
- SizeMode property of class PictureBox **36**, 424
- sizing handle **31**
- .sln file extension 36
- small circles in the UML **97**
- SmallImageList property of class ListView **475**
- smart tag menu **652**
- smartphone 2
- snap lines **409**, **410**
- Software Engineering
  - Observations overview xxviii
- software reuse 300, 493
- solid circle in the UML **98**
- solid circle surrounded by a hollow circle in the UML **98**
- SoLidBrush class **433**
- solution 11, **18**
- Solution Explorer** window **25**
- Sort method of class Array **596**
- Sort method of class List<T> 257
- Sorted property of class
  - ComboBox 465
- Sorted property of class ListBox 458
- SortedDictionary generic class 592, **599**, 601
- SortedDictionary<K, V> class 592
  - ContainsKey method **602**
  - Count property **602**
  - method Add **602**
  - property Values **603**
- SortedList class 592, 593
- SortedList<K, V> generic class **592**
- SortedSet<T> class **627**
- source code 41
- Source property of Exception 383
- space character 43
- space/time trade-off **599**
- spacing convention 45
- special character 57, **504**
- Split method of class Regex 601
- Split method of class String **544**
- SQL 246, **630**
- SQL Server Express 641
- SQL Server Express LocalDB **630**
- Sqrt method of class Math 388
- Sqrt method of Math 153, 154, 160
- square brackets, [] **197**
- square root 154
- stack **177**, 577
- Stack class 592, 593
- stack frame **177**
- Stack generic class 577, 592
  - Stack< double> 587
  - Stack<int> 587
- stack overflow **178**
- stack trace **365**
- stack unwinding 383
- Stack unwinding and Exception class properties 383
- Stack<T> class **592**
- StackOverflowException class **372**
- StackTrace property of Exception **382**, 383, 386
- standard error stream object **531**
- standard input stream object **531**
- standard input/output object **46**
- standard output stream object **531**
- standard reusable component **301**
- standard time format 268
- Start method of class Process **456**
- Start Page** **17**
- start tag **692**
- StartsWith and EndsWith methods 510
- StartsWith method of class string **263**, 510, 511
- Startup object** for a Visual Studio project 155
- startup project **25**
- state button **416**
- statement **46**, 74, 85
  - break 136, 141
  - continue **141**
  - control statement 97, 99, 100
  - control-statement nesting **99**
  - control-statement stacking **99**
  - do...while 98, 132, 133
  - double selection **98**
  - empty 103
  - for 98, 125, 128, 129
  - foreach **203**
  - if 61, 64, 98, 99, 100, 133
  - if...else 98, 100, 101, 112, 133
  - iteration **97**, **99**, 106
  - multiple selection **98**
  - nested **114**
  - nested if...else **101**
  - return 160
  - selection **97**, 98
  - single selection **98**
  - switch 98, 133, 138
  - switch multiple-selection statement 168
  - throw **268**, **379**
  - try **211**, **371**
  - using 381
  - while 98, 107, 109, 112
- statement lambda **614**
- static
  - class member 285
  - method **130**
  - variable **284**, 285
- static binding **351**
- static class 297
- static keyword 157
- static member demonstration 287
- static method 157
- static method cannot access non-static class members 285

- static method Concat 515
- static variable 154
- static variable scope 285
- static variable used to maintain
  - a count of the number of Employee objects in memory 286
- stereotype in the UML **83**
- straight-line form **60**
- stream
  - standard error **531**
  - standard input **531**
  - standard output **531**
- Stream class **531**
- stream of bytes **530**
- StreamReader class **531**
  - ReadToEnd method **561**
- StreamWriter class **531**
- StretchImage value 36
- string class **46, 503**
  - Concat method 515
  - constant **504**
  - CopyTo method 506
  - EndsWith method 510, 511
  - Equals method 508, 509
  - immutable 506
  - IndexOf method 511, 513
  - IndexOfAny method 511
  - LastIndexOf method 511, 513
  - LastIndexOfAny method 511, 513
  - Length property 506, 507
  - literal **46**
  - method ToLower 602
  - method ToUpper 606
  - Replace method 515, 516
  - Split method **544**
  - StartsWith method **263, 511**
  - Substring method 514
  - ToLower method 515, 516
  - ToUpper method **263, 515, 516**
  - Trim method 515, 517
  - verbatim **456, 504**
- String Collection Editor in Visual Studio .NET 459
- string concatenation **157, 286**
- string constructors 505
  - string format specifiers 92
  - string indexer 507
  - string indexer, Length property and CopyTo method 506
  - string interpolation (C# 6) **55, 56**
    - \$ **56**
  - string literal **504**
  - string type **55, 73**
  - string.Empty **190**
- StringBuilder class 503, 517
  - Append method 520
  - AppendFormat method 521
  - Capacity property 518
  - constructors 517
  - EnsureCapacity method 518
  - Length property 518
  - Remove method 522
  - Replace method 524
  - ToString method 517
- StringBuilder constructors 517
- StringBuilder size
  - manipulation 518
- StringBuilder text replacement 524
- struct
  - cannot define parameterless constructor 293
  - DateTime **499**
  - default constructor 293
- struct keyword **292**
- structured programming **97, 590**
- Structured Query Language (SQL) **630**
- Style property of class Font **418**
- submenu 439
- Substring method of class
  - string 514
- Subtract method of DateTime **625**
- subtraction 60
- Sum LINQ extension method **619**
- summarizing responses to a survey 208
- summing integers with the for statement 128
- switch code snippet (IDE) 174
- switch expression **133, 136**
- switch logic 139
- switch multiple-selection
  - statement 98, 133, 138, 168
    - activity diagram with break statements 138
    - case label 136, 137
    - default label **136, 168**
- Sybase 630
- synchronous programming 7
- syntax **42**
- syntax color highlighting **48**
- syntax error **42**
- syntax error underlining 52
- System 189
- System namespace 43, 164, 503
- System.Collections namespace 163, 574, **590**
- System.Collections.Concurrent namespace **590**
- System.Collections.Generic namespace 163, 256, 565, **590**
- System.Collections.Specialized namespace **590**
- System.Data.Entity namespace 163, **637**
- System.Diagnostics namespace **456**
- System.Drawing namespace 418
- System.IO namespace 163, 531
- System.Linq namespace 163, 248, 637
- System.Net.Http namespace 691
- System.Numerics namespace
  - BigInteger struct 189
- System.Runtime.Serialization
  - .Formatters.Binary namespace **550**
  - System.Runtime.Serialization.Json namespace 550
- System.Text namespace 163, 503
- System.Text.RegularExpressions namespace 503
- System.Threading.Tasks namespace **679**
- System.Web namespace 163
- System.Windows.Controls namespace 163



- System.Windows.Forms namespace 163, 397
  - System.Windows.Input namespace 163
  - System.Windows.Media namespace 163
  - System.Windows.Shapes namespace 163
  - System.Xml namespace 163
  - System.Xml.Linq namespace 163, **693**
  - System.Xml.Serialization namespace 550
  - SystemException class **372**, 388
- T**
- tab 396
  - tab character, \t 43, 54
  - tab stops 54
  - Tabbed pages in Visual Studio .NET 480
  - tabbed window 21
  - TabControl class **480**
    - ImageList property 481
    - ItemSize property 481
    - Multiline property 481
    - SelectedIndex property 481
    - SelectedIndexChanged event 481
    - SelectedTab property 481
    - TabCount property 481
    - TabPage property **480**, 481
  - TabControl with TabPages example 481
  - TabControl, adding a TabPage 481
  - TabCount property of class
    - TabControl 481
  - TabIndex property of class
    - Control **407**
  - table 225
  - table element 225
  - table in a relational database 631
  - table of simple types 702
  - table of values **225**
  - TabPage class **480**
    - add to TabControl 480, 481
    - Text property **480**
    - using radio buttons 484
  - TabPage property of class
    - TabControl **480**, 481
  - TabStop property of class
    - Control 407
  - tabular format 200
  - tagging 687
  - Tan method of Math 153
  - tangent 153
  - tape 530
  - TargetSite property of
    - Exception 383
  - Task class
    - Result property **679**
    - Run method **679**, 685
    - WhenAll method **685**
    - WhenAny method **686**
  - Task Parallel Library 679
  - Task<TResult> class **679**
  - Team menu 21
  - template **19**
  - temporary data storage 530
  - temporary value 113
  - termination housekeeping **284**
  - termination model of exception handling **371**
  - ternary operator **103**
  - test harness **224**
  - Testing class
    - BasePlusCommissionEmployee 312
  - Testing class
    - CommissionEmployee 308
  - Testing generic class Stack 581, 585
  - Tests interface IPayable with disparate classes 359
  - text editor 46, 503
  - Text property **30**, 33
  - Text property of class Button 411
  - Text property of class CheckBox 416
  - Text property of class Control 407
  - Text property of class Form 398
  - Text property of class GroupBox 413
  - Text property of class LinkLabel 454
  - Text property of class
    - RadioButton 419
  - Text property of class TabPage **480**
  - Text property of class TextBox 411
  - Text property of class
    - ToolStripMenuItem 443
  - Text property of class TreeNode 470
  - TextAlign property of a Label **34**
  - textbox **410**
  - TextBox control 396, 410
    - AcceptsReturn property 411
    - Multiline property 411
    - ReadOnly property 411
    - ScrollBars property 411
    - Text property 411
    - TextChanged event 411
    - UseSystemPasswordChar property **410**
  - TextChanged event of class
    - TextBox 411
  - Text-displaying application 42
  - TextReader class 531
  - TextWriter class 531
  - ThenBy extension method of class Queryable **650**
  - this
    - keyword 271, 272, 285
    - reference **271**
    - to call another constructor of the same class 276
  - this used implicitly and explicitly to refer to members of an object 271
  - thread
    - of execution **674**
  - ThreeState property of class
    - CheckBox **416**
  - throw an exception **211**, **268**, 276, **364**, 369
  - throw point **366**, 371
  - throw statement **379**
  - Tick event of class Timer **499**
  - tile 9
  - tiled window **487**
  - TileHorizontal value of enumeration MdiLayout 487

- TileVertical value of
  - enumeration MdiLayout 487
- time and date 499
- Time value of enumeration
  - DateTimePickerFormat **450**
- Time1 class declaration
  - maintains the time in 24-hour format 267
- Time1 object used in an app 269
- Time2 class declaration with
  - overloaded constructors 273
- TimeOfDay property of DateTime **450**
- Timer class **499**
  - Interval property **499**
  - Tick event **499**
- TimeSpan **625**
  - TotalMilliseconds property **625**
- TimeSpan value type 680
- title bar 30
- title bar, MDI parent and child 487
- Titles table of Books database 632, 633
- ToArray method of class
  - Enumerable 263
- tokenize a string 544
- ToList method of class
  - Enumerable 263, **625**
- ToLongDateString method of structure DateTime **453**
- ToLongTimeString method of structure DateTime **499**
- ToLower method of class string 515, 516, 602
- ToLower method of struct Char 527
- tool bar 396
- tool tip **23**
- toolbar **22**
- toolbar icon **22**
- Toolbox 26
- Tools menu 21
- ToolStripMenuItem class **440**
  - Checked property 443, **448**
  - OnClick property 443
  - Click event **442**, 443
  - ShortcutKeyDisplayString property **440**, 443
  - ShortcutKeys property **440**, 443
  - ShowShortcutKeys property **440**, 443
  - Text property 443
- ToolStripMenuItem properties and an event 443
- ToolTip class **426**
  - AutoPopDelay property 427
  - Draw event 427
  - InitialDelay property 427
  - ReshowDelay property 427
- ToolTip properties and events 427
- ToString method of an anonymous type 659
- ToString method of class
  - Exception 386
- ToString method of class
  - object 307, 326
- ToString method of class
  - StringBuilder 517, 520
- TotalMilliseconds property of TimeSpan **625**
- ToUpper method of class string **263**, 515, 516, 606
- ToUpper method of struct Char 527
- trace **584**
- transfer of control **96**
- transition arrow in the UML **97**, **98**, 100, 107
- traverse an array **227**
- tree **468**
- TreeNode class **469**
  - Checked property 469
  - Collapse method 470
  - Expand method 470
  - ExpandAll method 470
  - FirstNode property 469
  - FullPath property 469
  - GetNodeCount method 470
  - ImageIndex property 469
  - LastNode property 469
  - NextNode property 469
  - Nodes property 469
  - PrevNode property 470
  - SelectedImageIndex property 470
  - Text property 470
- TreeNode Editor 470
- TreeNode properties and methods 469
- TreeNodeCollection class **469**
- TreeView class 439, **468**
  - AfterSelected event 469
  - CheckBoxes property 469
  - ImageList property 469
  - Nodes property 469
  - SelectedNode property 469
- TreeView displaying a sample tree 468
- TreeView properties and an event 469
- TreeView used to display directories 471
- TreeViewEventArgs class **469**
  - trigger an event 396
- trigonometric cosine 153
- trigonometric sine 153
- trigonometric tangent 153
- Trim method of class string 515
- TrimExcess method of class
  - List<T> 257
- true 61, **99**, 100
- truncate 60, **110**
- truth table **144**
  - for operator ^ 145
  - for operator ! 146
  - for operator && 144
  - for operator || 144
- try block **211**, **369**
- try statement **211**, **371**
- TryParse method of structure
  - int **369**
- 24-hour clock format 267
- two-dimensional array **225**
- type **55**, 57
- type argument **572**, 573, 581
- type checking **568**
- Type class 326, **350**
  - FullName property 326
- type constraint **574**, 576
  - specifying **574**
- type inference **206**, 573
- type parameter **572**, 578, 587
  - scope 579
- type parameter list **572**, 578
- typesetting system 503
- typing in a TextBox 398

**U**

uint simple type 702  
ulong simple type 702  
UML  
  activity diagram 133  
UML (Unified Modeling Language) 5  
  activity diagram 97, 98, 100, 106  
  arrow 98  
  class diagram **76**, 83  
  compartment in a class diagram 76  
  diamond 99  
  dotted line **98**  
  final state **98**  
  guard condition **99**  
  merge symbol **106**  
  modeling properties 83  
  note **98**  
  solid circle **98**  
  solid circle surrounded by a hollow circle 98  
  stereotype **83**  
UML class diagram 301  
unary cast operator 113  
unary operator **114**, 146  
UnauthorizedAccessException class 473  
unboxing conversion 592  
uncaught exception **370**  
uneditable text or icons 396  
unhandled exception 366, **370**  
Unicode character set 121, 138, **504**  
Unified Modeling Language (UML) 5  
universal-time format 267, 268  
Universal Windows Platform (UWP) **10**  
unmanaged resource 361  
unqualified name 162, 174, 493  
unwind a method from the call stack 386  
UpDownAlign property of class NumericUpDown 429  
uppercase letter 44, 57  
UseMnemonic property of class LinkLabel 454  
user-defined classes 44

UserControl control **497**  
UserControl defined clock 498  
user-defined exception class **386**  
user-interface thread 622  
UseSystemPasswordChar property of class TextBox **410**  
ushort simple type 702  
using directive **43**, 162  
Using lambda expressions 612  
using static directive **595**

**V**

valid identifier 55  
validate data 78  
validate input 369  
validation **89**  
validity checking **89**  
value contextual keyword **82**  
Value property of a nullable type **391**  
Value property of class DateTimePicker **450**, 451, **452**  
Value property of class LinkedListNode **603**  
Value property of class NumericUpDown 429  
Value property of class XAttribute **693**  
value type **189**  
value type constraint struct **576**  
value types 292  
ValueChanged event  
  of class DateTimePicker **450**  
  of class NumericUpDown 429  
Values property of class SortedDictionary<K, V> **603**  
ValueType class **525**  
var keyword **206**  
variable 55  
  declaration statement **55**, 57  
  name **55**  
  variable is not modifiable 288  
  variable-length argument list **236**  
  variable scope 127  
  verbatim string **456**, **504**  
  syntax(@) 504  
VIEW menu 21, 24

View property of class ListView **474**, 474  
virtual  
  keyword **316**  
virtual machine (VM) **7**  
Visible property of class Control 407  
VisitedLinkColor property of class LinkLabel 454  
visual app development **16**  
visual programming 397  
Visual Studio **10**  
  component tray **427**  
  IDE (integrated development environment) 10  
  themes 17  
Visual Studio .NET Class View 289  
Visual Studio .NET Object Browser 289  
Visual Studio Community 2015 47  
Visual Studio® **16**  
void keyword 46, **73**

**W**

web service **687**  
when clause of a catch handler (C# 6) 392  
WhenAll method of class Task **685**  
WhenAny method of class Task **686**  
where clause **576**  
  of a LINQ query **250**  
Where extension method **621**  
  of class IQueryable **655**  
while iteration statement 98, 107  
  activity diagram in the UML 107  
while keyword 132  
while repetition statement 109, 112  
whitespace **43**, 46, 65  
  characters **43**  
whitespace character (regular expressions) 517  
whole-number literal 130

- widget **396**
  - Width property of structure Size **409**
  - window auto hide **24**
  - window gadget **396**
  - Window menu 22
  - window tab 21
  - Windows
    - Font 34
    - Properties **26**, 28, 30, 34
    - Solution Explorer **25**
  - Windows 10 10
  - Windows 8 9
  - Windows 8 UI 9
  - Windows bitmap (BMP) 36
  - Windows Explorer 456
  - Windows Forms **396**
  - Windows operating system **9**
  - Windows Phone 7 9
  - Windows Phone operating system 9
  - Windows Store xxvii, 10
  - word processor 503, 511
  - workflow **97**
  - Write method of class Console **53**, 531
  - WriteLine method of class Console 46, 53, 531
  - WriteLine method of class StreamWriter **539**
  - www.deitel.com/LINQ/ 264
- X**
- X format specifier 92
  - X property of class MouseEventArgs 431
  - XAttribute class **693**
    - Value property **693**
  - XDocument class **693**
    - Descendants method **693**
    - Parse method **693**
  - XElement class **693**
    - Attribute method **693**
  - XML (Extensible Markup Language) 687
    - element **692**
    - end tag **692**
    - start tag **692**
- Y**
- Y property of class MouseEventArgs 431
- Z**
- zeroth element **197**