The Complete Reference

Schildt's classic reference—fully revised and updated for C# 3.0

C# 3.0

"For answers about using C# in the real world, just pull this off the bookshelf. Indispensable!"
—*Michael Howard, Microsoft*

- Comprehensive coverage of the entire C# language
- Covers the new 3.0 features, including LINQ, lambda expressions, and anonymous types
- Includes hundreds of easy-to-understand examples

Herbert Schildt
Top-selling programming author with millions of books sold worldwide

McGraw Hill

C# 3.0:
The Complete Reference

About the Author

Herbert Schildt is a leading authority on C#, C++, C, and Java. His programming books have sold more than 3.5 million copies worldwide and have been translated into all major foreign languages. He is the author of numerous bestsellers, including *Java: The Complete Reference, C++: The Complete Reference, C: The Complete Reference,* and *C#: A Beginner's Guide.* Although interested in all facets of computing, his primary focus is computer languages, including compilers, interpreters, and robotic control languages. He also has an active interest in the standardization of languages. Schildt holds both graduate and undergraduate degrees from the University of Illinois. He can be reached at his consulting office at (217) 586-4683. His web site is **www.HerbSchildt.com**.

About the Technical Editor

Michael Howard (Austin, Texas) is a principal security program manager on the Trustworthy Computing (TwC) Group's Security Engineering team at Microsoft, where he is responsible for managing secure design, programming, and testing techniques across the company. Howard is an architect of the Security Development Lifecycle (SDL), a process for improving the security of Microsoft's software. Howard speaks regularly on the topic of securing code for Microsoft and at conferences worldwide. He regularly publishes articles on security design and is the co-author of six security books, including the award-winning *Writing Secure Code, 19 Deadly Sins of Software Security, The Security Development Lifecycle,* and his most recent release, *Writing Secure Code for Windows Vista.*

C# 3.0: The Complete Reference

Herbert Schildt

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

The McGraw-Hill Companies

Copyright © 2009 by The McGraw-Hill Companies. All rights reserved. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

ISBN: 978-0-07-159842-2

MHID: 0-07-159842-6

The material in this eBook also appears in the print version of this title: ISBN: 978-0-07-158841-6, MHID: 0-07-158841-8.

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

McGraw-Hill eBooks are available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. To contact a representative please visit the Contact Us page at www.mhprofessional.com.

Information has been obtained by McGraw-Hill from sources believed to be reliable. However, because of the possibility of human or mechanical error by our sources, McGraw-Hill, or others, McGraw-Hill does not guarantee the accuracy, adequacy, or completeness of any information and is not responsible for any errors or omissions or the results obtained from the use of such information.

TERMS OF USE

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

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

Contents at a Glance

Part I	**The C# Language**	
1	The Creation of C#	3
2	An Overview of C#	11
3	Data Types, Literals, and Variables	35
4	Operators	63
5	Program Control Statements	85
6	Introducing Classes and Objects	109
7	Arrays and Strings	137
8	A Closer Look at Methods and Classes	165
9	Operator Overloading	213
10	Indexers and Properties	245
11	Inheritance	269
12	Interfaces, Structures, and Enumerations	311
13	Exception Handling	337
14	Using I/O	363
15	Delegates, Events, and Lambda Expressions	399
16	Namespaces, the Preprocessor, and Assemblies	437
17	Runtime Type ID, Reflection, and Attributes	459
18	Generics	495
19	LINQ	545
20	Unsafe Code, Pointers, Nullable Types, and Miscellaneous Topics	585
Part II	**Exploring the C# Library**	
21	Exploring the System Namespace	615
22	Strings and Formatting	663
23	Multithreaded Programming	703
24	Collections, Enumerators, and Iterators	749
25	Networking Through the Internet Using System.Net	821
26	Use System.Windows.Forms to Create Form-Based Windows Applications	847
A	Documentation Comment Quick Reference	867
	Index	871

This page intentionally left blank

Contents

Special Thanks xxi
Preface xxiii

Part I **The C# Language**

1 **The Creation of C#** **3**
C#'s Family Tree 3
C: The Beginning of the Modern Age of Programming 3
The Creation of OOP and C++ 4
The Internet and Java Emerge 4
The Creation of C# 5
The Evolution of C# 7
How C# Relates to the .NET Framework 7
What Is the .NET Framework? 8
How the Common Language Runtime Works 8
Managed vs. Unmanaged Code 9
The Common Language Specification 9

2 **An Overview of C#** **11**
Object-Oriented Programming 11
Encapsulation 12
Polymorphism 12
Inheritance 13
A First Simple Program 13
Using csc.exe, the C# Command-Line Compiler 14
Using the Visual Studio IDE 15
The First Sample Program, Line by Line 19
Handling Syntax Errors 22
A Small Variation 22
A Second Simple Program 23
Another Data Type 25
Two Control Statements 26
The if Statement 27
The for Loop 28
Using Code Blocks 29
Semicolons, Positioning, and Indentation 31
The C# Keywords 32
Identifiers 33
The .NET Framework Class Library 34

3 Data Types, Literals, and Variables 35
Why Data Types Are Important 35
C#'s Value Types 35
Integers 36
Floating-Point Types 38
The decimal Type 40
Characters 41
The bool Type 42
Some Output Options 43
Literals 46
Hexadecimal Literals 47
Character Escape Sequences 47
String Literals 48
A Closer Look at Variables 49
Initializing a Variable 50
Dynamic Initialization 50
Implicitly Typed Variables 51
The Scope and Lifetime of Variables 52
Type Conversion and Casting 55
Automatic Conversions 55
Casting Incompatible Types 56
Type Conversion in Expressions 59
Using Casts in Expressions 60

4 Operators 63
Arithmetic Operators 63
Increment and Decrement 64
Relational and Logical Operators 67
Short-Circuit Logical Operators 70
The Assignment Operator 71
Compound Assignments 72
The Bitwise Operators 73
The Bitwise AND, OR, XOR, and NOT Operators 73
The Shift Operators 79
Bitwise Compound Assignments 81
The ? Operator 82
Spacing and Parentheses 83
Operator Precedence 84

5 Program Control Statements 85
The if Statement 85
Nested ifs 86
The if-else-if Ladder 87
The switch Statement 88
Nested switch Statements 92
The for Loop 92
Some Variations on the for Loop 94

The while Loop 99
The do-while Loop 101
The foreach Loop 102
Using break to Exit a Loop 102
Using continue 104
return 105
The goto 105

6 Introducing Classes and Objects 109
Class Fundamentals 109
The General Form of a Class 109
Define a Class 110
How Objects Are Created 114
Reference Variables and Assignment 115
Methods 116
Add a Method to the Building Class 117
Return from a Method 119
Return a Value 120
Use Parameters 122
Add a Parameterized Method to Building 125
Avoiding Unreachable Code 126
Constructors 126
Parameterized Constructors 128
Add a Constructor to the Building Class 129
The new Operator Revisited 130
Using new with Value Types 130
Garbage Collection and Destructors 131
Destructors 131
The this Keyword 133

7 Arrays and Strings 137
Arrays 137
One-Dimensional Arrays 137
Multidimensional Arrays 141
Two-Dimensional Arrays 141
Arrays of Three or More Dimensions 142
Initializing Multidimensional Arrays 143
Jagged Arrays 144
Assigning Array References 146
Using the Length Property 148
Using Length with Jagged Arrays 150
Implicitly Typed Arrays 151
The foreach Loop 152
Strings 156
Constructing Strings 156
Operating on Strings 157
Arrays of Strings 160

Strings Are Immutable 161
Strings Can Be Used in switch Statements 162

8 A Closer Look at Methods and Classes 165
Controlling Access to Class Members 165
C#'s Access Modifiers 165
Applying Public and Private Access 167
Controlling Access: A Case Study 168
Pass References to Methods 172
How Arguments Are Passed 174
Use ref and out Parameters 176
Use ref 177
Use out 178
Use ref and out on References 181
Use a Variable Number of Arguments 182
Return Objects 185
Return an Array 187
Method Overloading 188
Overload Constructors 194
Invoke an Overloaded Constructor Through this 197
Object Initializers 199
The Main( ) Method 200
Return Values from Main( ) 200
Pass Arguments to Main( ) 200
Recursion 202
Understanding static 205
Static Constructors 210
Static Classes 211

9 Operator Overloading 213
Operator Overloading Fundamentals 213
Overloading Binary Operators 214
Overloading Unary Operators 216
Handling Operations on C# Built-in Types 220
Overloading the Relational Operators 224
Overloading true and false 226
Overloading the Logical Operators 229
A Simple Approach to Overloading the Logical Operators 229
Enabling the Short-Circuit Operators 231
Conversion Operators 235
Operator Overloading Tips and Restrictions 239
Another Example of Operator Overloading 240

10 Indexers and Properties 245
Indexers 245
Creating One-Dimensional Indexers 245
Indexers Can Be Overloaded 249

Indexers Do Not Require an Underlying Array 251
Multidimensional Indexers 252
Properties 254
Auto-Implemented Properties 259
Use Object Initializers with Properties 260
Property Restrictions 261
Use Access Modifiers with Accessors 261
Using Indexers and Properties 264

11 Inheritance 269
Inheritance Basics 269
Member Access and Inheritance 272
Using Protected Access 275
Constructors and Inheritance 276
Calling Base Class Constructors 278
Inheritance and Name Hiding 282
Using base to Access a Hidden Name 283
Creating a Multilevel Hierarchy 285
When Are Constructors Called? 288
Base Class References and Derived Objects 289
Virtual Methods and Overriding 294
Why Overridden Methods? 297
Applying Virtual Methods 298
Using Abstract Classes 301
Using sealed to Prevent Inheritance 305
The object Class 305
Boxing and Unboxing 307
Is object a Universal Data Type? 309

12 Interfaces, Structures, and Enumerations 311
Interfaces 311
Implementing Interfaces 312
Using Interface References 316
Interface Properties 318
Interface Indexers 320
Interfaces Can Be Inherited 322
Name Hiding with Interface Inheritance 323
Explicit Implementations 323
Choosing Between an Interface and an Abstract Class 326
The .NET Standard Interfaces 326
Structures 326
Why Structures? 330
Enumerations 332
Initialize an Enumeration 333
Specify the Underlying Type of an Enumeration 334
Use Enumerations 334

13 Exception Handling 337
The System.Exception Class 337
Exception Handling Fundamentals 338
Using try and catch 338
A Simple Exception Example 338
A Second Exception Example 340
The Consequences of an Uncaught Exception 341
Exceptions Let You Handle Errors Gracefully 343
Using Multiple catch Clauses 344
Catching All Exceptions 345
Nesting try Blocks 346
Throwing an Exception 347
Rethrowing an Exception 348
Using finally 349
A Closer Look at the Exception Class 351
Commonly Used Exceptions 352
Deriving Exception Classes 354
Catching Derived Class Exceptions 358
Using checked and unchecked 360

14 Using I/O 363
C#'s I/O Is Built Upon Streams 363
Byte Streams and Character Streams 363
The Predefined Streams 363
The Stream Classes 364
The Stream Class 364
The Byte Stream Classes 365
The Character Stream Wrapper Classes 365
Binary Streams 367
Console I/O 367
Reading Console Input 367
Using ReadKey( ) 369
Writing Console Output 371
FileStream and Byte-Oriented File I/O 371
Opening and Closing a File 372
Reading Bytes from a FileStream 374
Writing to a File 375
Using FileStream to Copy a File 376
Character-Based File I/O 378
Using StreamWriter 378
Using a StreamReader 380
Redirecting the Standard Streams 381
Reading and Writing Binary Data 383
BinaryWriter 383
BinaryReader 384
Demonstrating Binary I/O 386

Random Access Files 390
Using MemoryStream 392
Using StringReader and StringWriter 394
Converting Numeric Strings to Their Internal Representation 396

15 Delegates, Events, and Lambda Expressions 399
Delegates 399
Delegate Method Group Conversion 402
Using Instance Methods as Delegates 402
Multicasting 404
Covariance and Contravariance 406
System.Delegate 408
Why Delegates 408
Anonymous Functions 408
Anonymous Methods 409
Pass Arguments to an Anonymous Method 410
Return a Value from an Anonymous Method 410
Use Outer Variables with Anonymous Methods 412
Lambda Expressions 413
The Lambda Operator 413
Expression Lambdas 414
Statement Lambdas 416
Events 419
A Multicast Event Example 421
Instance Methods vs. Static Methods as Event Handlers 422
Using Event Accessors 424
Miscellaneous Event Features 429
Use Anonymous Methods and Lambda Expressions with Events 429
.NET Event Guidelines 430
Use EventHandler 432
Applying Events: A Case Study 433

16 Namespaces, the Preprocessor, and Assemblies 437
Namespaces 437
Declaring a Namespace 438
Namespaces Prevent Name Conflicts 440
using 441
A Second Form of using 443
Namespaces Are Additive 444
Namespaces Can Be Nested 446
The Global Namespace 447
Using the :: Namespace Alias Qualifier 447
The Preprocessor 451
#define 452
#if and #endif 452
#else and #elif 454
#undef 455

#error ... 456
#warning ... 456
#line ... 456
#region and #endregion ... 456
#pragma ... 457
Assemblies and the internal Access Modifier ... 457
The internal Access Modifier ... 458

17 Runtime Type ID, Reflection, and Attributes ... 459
Runtime Type Identification ... 459
Testing a Type with is ... 459
Using as ... 460
Using typeof ... 462
Reflection ... 463
The Reflection Core: System.Type ... 463
Using Reflection ... 465
Obtaining Information About Methods ... 465
Calling Methods Using Reflection ... 469
Obtaining a Type's Constructors ... 471
Obtaining Types from Assemblies ... 475
Fully Automating Type Discovery ... 481
Attributes ... 483
Attribute Basics ... 483
Positional vs. Named Parameters ... 487
Three Built-in Attributes ... 491
AttributeUsage ... 491
The Conditional Attribute ... 491
The Obsolete Attribute ... 493

18 Generics ... 495
What Are Generics? ... 495
A Simple Generics Example ... 496
Generic Types Differ Based on Their Type Arguments ... 499
How Generics Improve Type Safety ... 499
A Generic Class with Two Type Parameters ... 502
The General Form of a Generic Class ... 503
Constrained Types ... 503
Using a Base Class Constraint ... 504
Using an Interface Constraint ... 512
Using the new( ) Constructor Constraint ... 516
The Reference Type and Value Type Constraints ... 517
Using a Constraint to Establish a Relationship Between Two Type Parameters ... 520
Using Multiple Constraints ... 521
Creating a Default Value of a Type Parameter ... 522
Generic Structures ... 523
Creating a Generic Method ... 524

Using Explicit Type Arguments to Call a Generic Method 527
Using a Constraint with a Generic Method 527
Generic Delegates 527
Generic Interfaces 530
Comparing Instances of a Type Parameter 534
Generic Class Hierarchies 537
Using a Generic Base Class 537
A Generic Derived Class 539
Overriding Virtual Methods in a Generic Class 540
Overloading Methods That Use Type Parameters 542
How Generic Types Are Instantiated 543
Some Generic Restrictions 544
Final Thoughts on Generics 544

19 LINQ 545
What Is LINQ? 545
LINQ Fundamentals 546
A Simple Query 546
A Query Can Be Executed More Than Once 548
How the Data Types in a Query Relate 549
The General Form of a Query 550
Filter Values with where 551
Sort Results with orderby 552
A Closer Look at select 556
Use Nested from Clauses 560
Group Results with group 561
Use into to Create a Continuation 563
Use let to Create a Variable in a Query 565
Join Two Sequences with join 566
Anonymous Types 569
Create a Group Join 571
The Query Methods 574
The Basic Query Methods 574
Create Queries by Using the Query Methods 575
Query Syntax vs. Query Methods 577
More Query-Related Extension Methods 577
Deferred vs. Immediate Query Execution 580
Expression Trees 581
Extension Methods 582

20 Unsafe Code, Pointers, Nullable Types, and Miscellaneous Topics 585
Unsafe Code 585
Pointer Basics 586
Using unsafe 587
Using fixed 588
Accessing Structure Members Through a Pointer 589

Pointer Arithmetic . . . 589
Pointer Comparisons . . . 591
Pointers and Arrays . . . 591
Pointers and Strings . . . 593
Multiple Indirection . . . 594
Arrays of Pointers . . . 595
stackalloc . . . 596
Creating Fixed-Size Buffers . . . 596
Nullable Types . . . 598
Nullable Basics . . . 598
Nullable Objects in Expressions . . . 600
The ?? Operator . . . 601
Nullable Objects and the Relational and Logical Operators . . . 602
Partial Types . . . 603
Partial Methods . . . 604
Friend Assemblies . . . 605
Miscellaneous Keywords . . . 605
lock . . . 605
readonly . . . 606
const and volatile . . . 607
The using Statement . . . 607
extern . . . 608

Part II **Exploring the C# Library**

21 **Exploring the System Namespace . . . 615**
The Members of System . . . 615
The Math Class . . . 617
The .NET Structures Corresponding to the Built-in Value Types . . . 623
The Integer Structures . . . 623
The Floating-Point Structures . . . 626
Decimal . . . 630
Char . . . 634
The Boolean Structure . . . 640
The Array Class . . . 641
Sorting and Searching Arrays . . . 648
Reversing an Array . . . 650
Copying an Array . . . 651
Using a Predicate . . . 652
Using an Action . . . 653
BitConverter . . . 654
Generating Random Numbers with Random . . . 656
Memory Management and the GC Class . . . 657
Object . . . 659
The IComparable and IComparable<T> Interfaces . . . 659
The IEquatable<T> Interface . . . 660

The IConvertible Interface 660
The ICloneable Interface 660
IFormatProvider and IFormattable 662

22 Strings and Formatting 663
Strings in C# 663
The String Class 664
The String Constructors 664
The String Field, Indexer, and Property 665
The String Operators 665
The String Methods 665
Padding and Trimming Strings 681
Inserting, Removing, and Replacing 682
Changing Case 683
Using the Substring( ) Method 684
The String Extension Methods 684
Formatting 684
Formatting Overview 685
The Numeric Format Specifiers 686
Understanding Argument Numbers 687
Using String.Format( ) and ToString( ) to Format Data 688
Using String.Format( ) to Format Values 688
Using ToString( ) to Format Data 691
Creating a Custom Numeric Format 692
The Custom Format Placeholder Characters 692
Formatting Date and Time 695
Creating a Custom Date and Time Format 698
Formatting Enumerations 700

23 Multithreaded Programming 703
Multithreading Fundamentals 703
The Thread Class 704
Creating and Starting a Thread 704
Some Simple Improvements 707
Creating Multiple Threads 708
Determining When a Thread Ends 710
Passing an Argument to a Thread 713
The IsBackground Property 715
Thread Priorities 715
Synchronization 717
An Alternative Approach 721
The Monitor Class and lock 723
Thread Communication Using Wait( ), Pulse( ), and PulseAll( ) 723
An Example That Uses Wait( ) and Pulse( ) 724
Deadlock and Race Conditions 727
Using MethodImplAttribute 728
Using a Mutex and a Semaphore 730

The Mutex 730
The Semaphore 734
Using Events 737
The Interlocked Class 739
Terminating a Thread 741
An Abort( ) Alternative 742
Canceling Abort( ) 743
Suspending and Resuming a Thread 745
Determining a Thread's State 745
Using the Main Thread 746
Multithreading Tips 747
Starting a Separate Task 747

24 Collections, Enumerators, and Iterators 749
Collections Overview 749
The Non-Generic Collections 750
The Non-Generic Interfaces 751
The DictionaryEntry Structure 755
The Non-Generic Collection Classes 755
Storing Bits with BitArray 771
The Specialized Collections 774
The Generic Collections 774
The Generic Interfaces 775
The KeyValuePair<TK, TV> Structure 778
The Generic Collection Classes 779
Storing User-Defined Classes in Collections 799
Implementing IComparable 801
Implementing IComparable for Non-Generic Collections 802
Implementing IComparable<T> for Generic Collections 803
Using an IComparer 805
Using a Non-Generic IComparer 805
Using a Generic IComparer<T> 806
Accessing a Collection via an Enumerator 808
Using an Enumerator 808
Using the IDictionaryEnumerator 809
Implementing IEnumerable and IEnumerator 811
Using Iterators 813
Stopping an Iterator 815
Using Multiple yield Directives 815
Creating a Named Iterator 816
Creating a Generic Iterator 818
Collection Initializers 819

25 Networking Through the Internet Using System.Net 821
The System.Net Members 821
Uniform Resource Identifiers 823

Internet Access Fundamentals 823
WebRequest 824
WebResponse 826
HttpWebRequest and HttpWebResponse 826
A Simple First Example 827
Handling Network Errors 830
Exceptions Generated by Create( ) 830
Exceptions Generated by GetReponse( ) 830
Exceptions Generated by GetResponseStream( ) 831
Using Exception Handling 831
The Uri Class 833
Accessing Additional HTTP Response Information 834
Accessing the Header 834
Accessing Cookies 836
Using the LastModified Property 838
MiniCrawler: A Case Study 839
Using WebClient 842

26 Use System.Windows.Forms to Create Form-Based Windows Applications 847
A Brief History of Windows Programming 847
Two Ways to Write a Form-Based Windows Application 848
How Windows Interacts with the User 848
Windows Forms 849
The Form Class 849
A Skeletal Form-Based Windows Program 849
Compiling the Windows Skeleton 851
Adding a Button 852
Button Basics 852
Adding a Button to a Form 852
A Simple Button Example 853
Handling Messages 853
An Alternative Implementation 856
Using a Message Box 856
Adding a Menu 859
Creating a Traditional-Style Main Menu 859
Creating a New-Style Menu with MenuStrip 863

A Documentation Comment Quick Reference 867
The XML Comment Tags 867
Compiling Documentation Comments 868
An XML Documentation Example 869

Index 871

This page intentionally left blank

Special Thanks

Special thanks go to Michael Howard for his excellent technical edit of this book. His expertise, insights, suggestions, and advice were of great value.

This page intentionally left blank

Preface

We programmers are a demanding bunch, always looking for ways to improve the performance, efficiency, and portability of our programs. We also demand much from the tools we use, especially when it comes to programming languages. There are many programming languages, but only a few are great. A great programming language must be powerful, yet flexible. Its syntax must be terse, but clear. It must facilitate the creation of correct code while not getting in our way. It must support state-of-the-art features, but not trendy dead ends. Finally, a great programming language must have one more, almost intangible quality: It must feel right when we use it. C# is such a language.

Created by Microsoft to support its .NET Framework, C# builds on a rich programming heritage. Its chief architect was long-time programming guru Anders Hejlsberg. C# is directly descended from two of the world's most successful computer languages: C and C++. From C, it derives its syntax, many of its keywords, and its operators. It builds upon and improves the object model defined by C++. C# is also closely related to another very successful language: Java.

Sharing a common ancestry, but differing in many important ways, C# and Java are more like cousins. Both support distributed programming and both use intermediate code to achieve safety and portability, but the details differ. They both also provide a significant amount of runtime error checking, security, and managed execution, but again, the details differ. However, unlike Java, C# also gives you access to pointers—a feature supported by C++. Thus, C# combines the raw power of C++ with the type safety of Java. Furthermore, the trade-offs between power and safety are carefully balanced and are nearly transparent.

Throughout the history of computing, programming languages have evolved to accommodate changes in the computing environment, advances in computer language theory, and new ways of thinking about and approaching the job of programming. C# is no exception. In the ongoing process of refinement, adaptation, and innovation, C# has demonstrated its ability to respond rapidly to the changing needs of the programmer. This fact is testified to by the many new features added to C# since its initial 1.0 release in 2000.

Consider the first major revision, C# 2.0. It added several features that made it easier for programmers to write more resilient, reliable, and nimble code. Without question, the most important 2.0 addition was generics. Through the use of generics, it became possible to create type-safe, reusable code in C#. Thus, the addition of generics fundamentally expanded the power and scope of the language.

Now consider the second major revision, C# 3.0. This is the latest version of C# and is the version described in this book. It is not an exaggeration to say that C# 3.0 has added features that have redefined the very core of C#, raising the bar in computer language development in the process. Of its many innovative features, two stand out: LINQ and lambda expressions. LINQ, which stands for Language Integrated Query, enables you to create database-style queries by using elements of the C# language. Lambda expressions implement a functional-style syntax that uses the => lambda operator, and lambda expressions are frequently used in LINQ expressions.

As you will see in the course of this book, the combination of LINQ and lambda expressions represents a radically powerful subset of C#. Furthermore, they are revolutionary features that

are redefining how solutions are crafted for many different types of programming tasks, not just database queries. In essence, they let you approach old problems in new ways. Their use not only streamlines a solution, but also helps you conceptualize a problem from a different point of view. Simply put, the addition of LINQ and lambda expressions is both significant and far reaching. They are changing the way we think about the job of programming.

Because of its ability to adapt rapidly to the changing demands of the programming landscape, C# has remained a vibrant and innovative language. As a result, it defines one of the most powerful, feature-rich languages in modern computing. It is also a language that no programmer can afford to ignore. This book is designed to help you master it.

What's Inside

This book describes C# 3.0. It is divided into two parts. Part I provides a comprehensive discussion of the C# language, including the new features added by version 3.0. This is the largest part in the book, and it describes the keywords, syntax, and features that define the language. I/O, file handling, reflection, and the preprocessor are also discussed in Part I.

Part II explores the C# class library, which is the .NET Framework class library. This library is huge! Because of space limitations, it is not possible to cover the entire .NET Framework class library in one book. Instead, Part II focuses on the core library, which is contained in the **System** namespace. Also covered are collections, multithreading, networking, and Windows Forms. These are the parts of the library that nearly every C# programmer will use.

A Book for All Programmers

This book does not require any previous programming experience. If you already know C++ or Java, you will be able to advance quite rapidly because C# has much in common with those languages. If you don't have any previous programming experience, you will still be able to learn C# from this book, but you will need to work carefully through the examples in each chapter.

Required Software

To compile and run C# 3.0 programs, you must use Visual Studio 2008 or later.

Don't Forget: Code on the Web

Remember, the source code for all of the programs in this book is available free-of-charge on the Web at **www.mhprofessional.com**.

For Further Study

C# 3.0: The Complete Reference is your gateway to the Herb Schildt series of programming books. Here are some others that you will find of interest.

For a carefully paced introduction to C#, try

C# 3.0: A Beginner's Guide

To learn about Java programming, we recommend the following:

Java: The Complete Reference

Java: A Beginner's Guide

Swing: A Beginner's Guide

The Art of Java

Herb Schildt's Java Programming Cookbook

To learn about C++, you will find these books especially helpful:

C++: The Complete Reference

C++: A Beginner's Guide

C++ From the Ground Up

STL Programming From the Ground Up

The Art of C++

Herb Schildt's C++ Programming Cookbook

If you want to learn about the C language, the foundation of all modern programming, the following title will be of interest:

C: The Complete Reference

When you need solid answers, fast, turn to Herbert Schildt, the recognized authority on programming.

This page intentionally left blank

PART I

The C# Language

Part I discusses the elements of the C# language, including its keywords, syntax, and operators. Also described are several foundational C# techniques, such as using I/O and reflection, which are tightly linked with the C# language.

CHAPTER 1
The Creation of C#
CHAPTER 2
An Overview of C#
CHAPTER 3
Data Types, Literals, and Variables
CHAPTER 4
Operators
CHAPTER 5
Program Control Statements
CHAPTER 6
Introducing Classes and Objects
CHAPTER 7
Arrays and Strings
CHAPTER 8
A Closer Look at Methods and Classes
CHAPTER 9
Operator Overloading
CHAPTER 10
Indexers and Properties
CHAPTER 11
Inheritance
CHAPTER 12
Interfaces, Structures, and Enumerations
CHAPTER 13
Exception Handling
CHAPTER 14
Using I/O
CHAPTER 15
Delegates, Events, and Lambda Expressions
CHAPTER 16
Namespaces, the Preprocessor, and Assemblies
CHAPTER 17
Runtime Type ID, Reflection, and Attributes
CHAPTER 18
Generics
CHAPTER 19
LINQ
CHAPTER 20
Unsafe Code, Pointers, Nullable Types, and Miscellaneous Topics

This page intentionally left blank

CHAPTER 1

The Creation of C#

C# is Microsoft's premier language for .NET development. It leverages time-tested features with cutting-edge innovations and provides a highly usable, efficient way to write programs for the modern enterprise computing environment. It is, by any measure, one of the most important languages of the 21st century.

The purpose of this chapter is to place C# into its historical context, including the forces that drove its creation, its design philosophy, and how it was influenced by other computer languages. This chapter also explains how C# relates to the .NET Framework. As you will see, C# and the .NET Framework work together to create a highly refined programming environment.

C#'s Family Tree

Computer languages do not exist in a void. Rather, they relate to one another, with each new language influenced in one form or another by the ones that came before. In a process akin to cross-pollination, features from one language are adapted by another, a new innovation is integrated into an existing context, or an older construct is removed. In this way, languages evolve and the art of programming advances. C# is no exception.

C# inherits a rich programming legacy. It is directly descended from two of the world's most successful computer languages: C and C++. It is closely related to another: Java. Understanding the nature of these relationships is crucial to understanding C#. Thus, we begin our examination of C# by placing it in the historical context of these three languages.

C: The Beginning of the Modern Age of Programming

The creation of C marks the beginning of the modern age of programming. C was invented by Dennis Ritchie in the 1970s on a DEC PDP-11 that used the UNIX operating system. While some earlier languages, most notably Pascal, had achieved significant success, C established the paradigm that still charts the course of programming today.

C grew out of the *structured programming* revolution of the 1960s. Prior to structured programming, large programs were difficult to write because the program logic tended to degenerate into what is known as "spaghetti code," a tangled mass of jumps, calls, and returns that is difficult to follow. Structured languages addressed this problem by adding well-defined control statements, subroutines with local variables, and other improvements. Through the use of structured techniques programs became better organized, more reliable, and easier to manage.

Although there were other structured languages at the time, C was the first to successfully combine power, elegance, and expressiveness. Its terse, yet easy-to-use syntax coupled with its philosophy that the programmer (not the language) was in charge quickly won many converts. It can be a bit hard to understand from today's perspective, but C was a breath of fresh air that programmers had long awaited. As a result, C became the most widely used structured programming language of the 1980s.

However, even the venerable C language had its limits. One of the most troublesome was its inability to handle large programs. The C language hits a barrier once a project reaches a certain size, and after that point, C programs are difficult to understand and maintain. Precisely where this limit is reached depends upon the program, the programmer, and the tools at hand, but there is always a threshold beyond which a C program becomes unmanageable.

The Creation of OOP and C++

By the late 1970s, the size of many projects was near or at the limits of what structured programming methodologies and the C language could handle. To solve this problem, a new way to program began to emerge. This method is called *object-oriented programming* (OOP). Using OOP, a programmer could handle much larger programs. The trouble was that C, the most popular language at the time, did not support object-oriented programming. The desire for an object-oriented version of C ultimately led to the creation of C++.

C++ was invented by Bjarne Stroustrup beginning in 1979 at Bell Laboratories in Murray Hill, New Jersey. He initially called the new language "C with Classes." However, in 1983 the name was changed to C++. C++ contains the entire C language. Thus, C is the foundation upon which C++ is built. Most of the additions that Stroustrup made to C were designed to support object-oriented programming. In essence, C++ is the object-oriented version of C. By building upon the foundation of C, Stroustrup provided a smooth migration path to OOP. Instead of having to learn an entirely new language, a C programmer needed to learn only a few new features before reaping the benefits of the object-oriented methodology.

C++ simmered in the background during much of the 1980s, undergoing extensive development. By the beginning of the 1990s, C++ was ready for mainstream use, and its popularity exploded. By the end of the decade, it had become the most widely used programming language. Today, C++ is still the preeminent language for the development of high-performance system code.

It is critical to understand that the invention of C++ was not an attempt to create an entirely new programming language. Instead, it was an enhancement to an already highly successful language. This approach to language development—beginning with an existing language and moving it forward—established a trend that continues today.

The Internet and Java Emerge

The next major advance in programming languages is Java. Work on Java, which was originally called Oak, began in 1991 at Sun Microsystems. The main driving force behind Java's design was James Gosling. Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan also played a role.

Java is a structured, object-oriented language with a syntax and philosophy derived from C++. The innovative aspects of Java were driven not so much by advances in the art of programming (although some certainly were), but rather by changes in the computing environment. Prior to the mainstreaming of the Internet, most programs were written,

compiled, and targeted for a specific CPU and a specific operating system. While it has always been true that programmers like to reuse their code, the ability to port a program easily from one environment to another took a backseat to more pressing problems. However, with the rise of the Internet, in which many different types of CPUs and operating systems are connected, the old problem of portability reemerged with a vengeance. To solve the problem of portability, a new language was needed, and this new language was Java.

Although the single most important aspect of Java (and the reason for its rapid acceptance) is its ability to create cross-platform, portable code, it is interesting to note that the original impetus for Java was not the Internet, but rather the need for a platform-independent language that could be used to create software for embedded controllers. In 1993, it became clear that the issues of cross-platform portability found when creating code for embedded controllers are also encountered when attempting to create code for the Internet. Remember: the Internet is a vast, distributed computing universe in which many different types of computers live. The same techniques that solved the portability problem on a small scale could be applied to the Internet on a large scale.

Java achieved portability by translating a program's source code into an intermediate language called *bytecode.* This bytecode was then executed by the Java Virtual Machine (JVM). Therefore, a Java program could run in any environment for which a JVM was available. Also, since the JVM is relatively easy to implement, it was readily available for a large number of environments.

Java's use of bytecode differed radically from both C and C++, which were nearly always compiled to executable machine code. Machine code is tied to a specific CPU and operating system. Thus, if you wanted to run a C/C++ program on a different system, it needed to be recompiled to machine code specifically for that environment. Therefore, to create a C/C++ program that would run in a variety of environments, several different executable versions of the program would be needed. Not only was this impractical, it was expensive. Java's use of an intermediate language was an elegant, cost-effective solution. It is also a solution that C# would adapt for its own purposes.

As mentioned, Java is descended from C and C++. Its syntax is based on C, and its object model is evolved from C++. Although Java code is neither upwardly nor downwardly compatible with C or C++, its syntax is sufficiently similar that the large pool of existing C/C++ programmers could move to Java with very little effort. Furthermore, because Java built upon and improved an existing paradigm, Gosling, et al., were free to focus their attention on the new and innovative features. Just as Stroustrup did not need to "reinvent the wheel" when creating C++, Gosling did not need to create an entirely new language when developing Java. Moreover, with the creation of Java, C and C++ became an accepted substrata upon which to base a new computer language.

The Creation of C#

While Java has successfully addressed many of the issues surrounding portability in the Internet environment, there are still features that it lacks. One is *cross-language interoperability,* also called *mixed-language programming.* This is the ability for the code produced by one language to work easily with the code produced by another. Cross-language interoperability is needed for the creation of large, distributed software systems. It is also desirable for programming software components because the most valuable component is one that can be used by the widest variety of computer languages, in the greatest number of operating environments.

Another feature lacking in Java is full integration with the Windows platform. Although Java programs can be executed in a Windows environment (assuming that the Java Virtual Machine has been installed), Java and Windows are not closely coupled. Since Windows is the most widely used operating system in the world, lack of direct support for Windows is a drawback to Java.

To answer these and other needs, Microsoft developed C#. C# was created at Microsoft late in the 1990s and was part of Microsoft's overall .NET strategy. It was first released in its alpha version in the middle of 2000. C#'s chief architect was Anders Hejlsberg. Hejlsberg is one of the world's leading language experts, with several notable accomplishments to his credit. For example, in the 1980s he was the original author of the highly successful and influential Turbo Pascal, whose streamlined implementation set the standard for all future compilers.

C# is directly related to C, C++, and Java. This is not by accident. These are three of the most widely used—and most widely liked—programming languages in the world. Furthermore, at the time of C#'s creation, nearly all professional programmers knew C, C++, and/or Java. By building C# upon a solid, well-understood foundation, C# offered an easy migration path from these languages. Since it was neither necessary nor desirable for Hejlsberg to "reinvent the wheel," he was free to focus on specific improvements and innovations.

The family tree for C# is shown in Figure 1-1. The grandfather of C# is C. From C, C# derives its syntax, many of its keywords, and its operators. C# builds upon and improves the object model defined by C++. If you know C or C++, then you will feel at home with C#.

C# and Java have a bit more complicated relationship. As explained, Java is also descended from C and C++. It too shares the C/C++ syntax and object model. Like Java, C# is designed to produce portable code. However, C# is not descended from Java. Instead, C# and Java are more like cousins, sharing a common ancestry, but differing in many important ways. The good news, though, is that if you know Java, then many C# concepts will be familiar. Conversely, if in the future you need to learn Java, then many of the things you learn about C# will carry over.

C# contains many innovative features that we will examine at length throughout the course of this book, but some of its most important relate to its built-in support for software components. In fact, C# has been characterized as being a component-oriented language because it contains integral support for the writing of software components. For example,

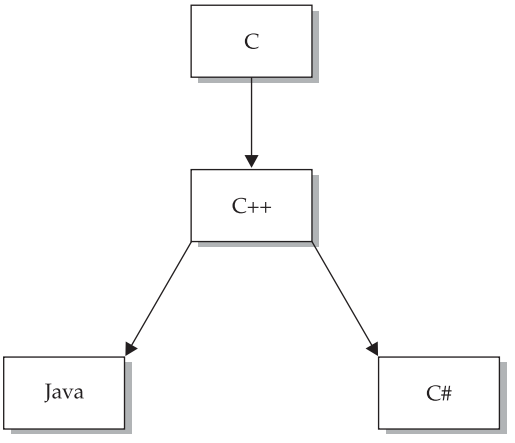


Figure 1-1
The C# family tree

C# includes features that directly support the constituents of components, such as properties, methods, and events. However, C#'s ability to work in a secure, mixed-language environment is perhaps its most important component-oriented feature.

The Evolution of C#

Since its original 1.0 release, C# has been evolving at a rapid pace. Not long after C# 1.0, Microsoft released version 1.1. It contained many minor tweaks but added no major features. However, the situation was much different with the release of C# 2.0.

C# 2.0 was a watershed event in the lifecycle of C# because it added many new features, such as generics, partial types, and anonymous methods, that fundamentally expanded the scope, power, and range of the language. Version 2.0 firmly put C# at the forefront of computer language development. It also demonstrated Microsoft's long-term commitment to the language.

The next major release of C# was 3.0, and this is the version of C# described by this book. Because of the many new features added by C# 2.0, one might have expected the development of C# to slow a bit, just to let programmers catch up, but this was not the case. With the release of C# 3.0, Microsoft once again put C# on the cutting edge of language design, this time adding a set of innovative features that redefined the programming landscape. Here is a list of what 3.0 has added to the language:

- Anonymous types
- Auto-implemented properties
- Extension methods
- Implicitly typed variables
- Lambda expressions
- Language-integrated query (LINQ)
- Object and collection initializers
- Partial methods

Although all of these features are important and have significant impact on the language, the two that are the most exciting are language-integrated query (LINQ) and lambda expressions. LINQ enables you to write database-style queries using C# programming elements. However, the LINQ syntax is not limited to only databases. It can also be used with arrays and collections. Thus, LINQ offers a new way to approach several common programming tasks. Lambda expressions are often used in LINQ expressions, but can also be used elsewhere. They implement a functional-style syntax that uses the lambda operator `=>`. Together, LINQ and lambda expressions add an entirely new dimension to C# programming. Throughout the course of this book, you will see how these features are revolutionizing the way that C# code is written.

How C# Relates to the .NET Framework

Although C# is a computer language that can be studied on its own, it has a special relationship to its runtime environment, the .NET Framework. The reason for this is twofold. First, C# was initially designed by Microsoft to create code for the .NET

Framework. Second, the libraries used by C# are the ones defined by the .NET Framework. Thus, even though it is theoretically possible to separate C# the language from the .NET environment, in practice the two are closely linked. Because of this, it is important to have a general understanding of the .NET Framework and why it is important to C#.

What Is the .NET Framework?

The .NET Framework defines an environment that supports the development and execution of highly distributed, component-based applications. It enables differing computer languages to work together and provides for security, program portability, and a common programming model for the Windows platform. As it relates to C#, the .NET Framework defines two very important entities. The first is the *Common Language Runtime* (CLR). This is the system that manages the execution of your program. Along with other benefits, the Common Language Runtime is the part of the .NET Framework that enables programs to be portable, supports mixed-language programming, and provides for secure execution.

The second entity is the .NET *class library.* This library gives your program access to the runtime environment. For example, if you want to perform I/O, such as displaying something on the screen, you will use the .NET class library to do it. If you are new to programming, then the term *class* may be new. Although it is explained in detail later in this book, for now a brief definition will suffice: a class is an object-oriented construct that helps organize programs. As long as your program restricts itself to the features defined by the .NET class library, your programs can run anywhere that the .NET runtime system is supported. Since C# automatically uses the .NET Framework class library, C# programs are automatically portable to all .NET environments.

How the Common Language Runtime Works

The Common Language Runtime manages the execution of .NET code. Here is how it works: When you compile a C# program, the output of the compiler is not executable code. Instead, it is a file that contains a special type of pseudocode called *Microsoft Intermediate Language* (MSIL). MSIL defines a set of portable instructions that are independent of any specific CPU. In essence, MSIL defines a portable assembly language. One other point: although MSIL is similar in concept to Java's bytecode, the two are not the same.

It is the job of the CLR to translate the intermediate code into executable code when a program is run. Thus, any program compiled to MSIL can be run in any environment for which the CLR is implemented. This is part of how the .NET Framework achieves portability.

Microsoft Intermediate Language is turned into executable code using a *JIT compiler*. "JIT" stands for "Just-In-Time." The process works like this: When a .NET program is executed, the CLR activates the JIT compiler. The JIT compiler converts MSIL into native code on demand as each part of your program is needed. Thus, your C# program actually executes as native code even though it is initially compiled into MSIL. This means that your program runs nearly as fast as it would if it had been compiled to native code in the first place, but it gains the portability benefits of MSIL.

In addition to MSIL, one other thing is output when you compile a C# program: *metadata*. Metadata describes the data used by your program and enables your code to interact easily with other code. The metadata is contained in the same file as the MSIL.

Managed vs. Unmanaged Code

In general, when you write a C# program, you are creating what is called *managed code.* Managed code is executed under the control of the Common Language Runtime as just described. Because it is running under the control of the CLR, managed code is subject to certain constraints—and derives several benefits. The constraints are easily described and met: the compiler must produce an MSIL file targeted for the CLR (which C# does) and use the .NET class library (which C# does). The benefits of managed code are many, including modern memory management, the ability to mix languages, better security, support for version control, and a clean way for software components to interact.

The opposite of managed code is unmanaged code. Unmanaged code does not execute under the Common Language Runtime. Thus, all Windows programs prior to the creation of the .NET Framework use unmanaged code. It is possible for managed code and unmanaged code to work together, so the fact that C# generates managed code does not restrict its ability to operate in conjunction with preexisting programs.

The Common Language Specification

Although all managed code gains the benefits provided by the CLR, if your code will be used by other programs written in different languages, then for maximum usability, it should adhere to the Common Language Specification (CLS). The CLS describes a set of features that different .NET-compatible languages have in common. CLS compliance is especially important when creating software components that will be used by other languages. The CLS includes a subset of the *Common Type System* (CTS). The CTS defines the rules concerning data types. Of course, C# supports both the CLS and the CTS.

This page intentionally left blank

CHAPTER 2

An Overview of C#

By far, the hardest thing about learning a programming language is the fact that no element exists in isolation. Instead, the components of the language work together. This interrelatedness makes it difficult to discuss one aspect of C# without involving another. To help overcome this problem, this chapter provides a brief overview of several C# features, including the general form of a C# program, some basic control statements, and operators. It does not go into too many details, but rather concentrates on the general concepts common to any C# program. Most of the topics discussed here are examined in greater detail in the remaining chapters of Part I.

Object-Oriented Programming

At the center of C# is *object-oriented programming* (OOP). The object-oriented methodology is inseparable from C#, and all C# programs are to at least some extent object oriented. Because of its importance to C#, it is useful to understand OOP's basic principles before you write even a simple C# program.

OOP is a powerful way to approach the job of programming. Programming methodologies have changed dramatically since the invention of the computer, primarily to accommodate the increasing complexity of programs. For example, when computers were first invented, programming was done by toggling in the binary machine instructions using the computer's front panel. As long as programs were just a few hundred instructions long, this approach worked. As programs grew, assembly language was invented so that a programmer could deal with larger, increasingly complex programs, using symbolic representations of the machine instructions. As programs continued to grow, high-level languages such as FORTRAN and COBOL were introduced that gave the programmer more tools with which to handle complexity. When these early languages began to reach their breaking point, structured programming languages, such as C, were invented.

At each milestone in the history of programming, techniques and tools were created to allow the programmer to deal with increasingly greater complexity. Each step of the way, the new approach took the best elements of the previous methods and moved forward. The same is true of object-oriented programming. Prior to OOP, many projects were nearing (or exceeding) the point where the structured approach no longer worked. A better way to handle complexity was needed, and object-oriented programming was the solution.

Object-oriented programming took the best ideas of structured programming and combined them with several new concepts. The result was a different and better way of organizing a program. In the most general sense, a program can be organized in one of two ways: around its code (what is happening) or around its data (what is being affected). Using only structured programming techniques, programs are typically organized around code. This approach can be thought of as "code acting on data."

Object-oriented programs work the other way around. They are organized around data, with the key principle being "data controlling access to code." In an object-oriented language, you define the data and the code that is permitted to act on that data. Thus, a data type defines precisely the operations that can be applied to that data.

To support the principles of object-oriented programming, all OOP languages, including C#, have three traits in common: encapsulation, polymorphism, and inheritance. Let's examine each.

Encapsulation

Encapsulation is a programming mechanism that binds together code and the data it manipulates, and that keeps both safe from outside interference and misuse. In an object-oriented language, code and data can be bound together in such a way that a self-contained *black box* is created. Within the box are all necessary data and code. When code and data are linked together in this fashion, an *object* is created. In other words, an object is the device that supports encapsulation.

Within an object, the code, data, or both may be *private* to that object or *public.* Private code or data is known to and accessible by only another part of the object. That is, private code or data cannot be accessed by a piece of the program that exists outside the object. When code or data is public, other parts of your program can access it even though it is defined within an object. Typically, the public parts of an object are used to provide a controlled interface to the private elements.

C#'s basic unit of encapsulation is the *class.* A class defines the form of an object. It specifies both the data and the code that will operate on that data. C# uses a class specification to construct *objects.* Objects are instances of a class. Thus, a class is essentially a set of plans that specify how to build an object.

Collectively, the code and data that constitute a class are called its *members*. The data defined by the class is referred to as *fields*. The terms *member variables* and *instance variables* also are used. The code that operates on that data is contained within *function members,* of which the most common is the *method.* Method is C#'s term for a subroutine. (Other function members include properties, events, and constructors.) Thus, the methods of a class contain code that acts on the fields defined by that class.

Polymorphism

Polymorphism (from Greek, meaning "many forms") is the quality that allows one interface to access a general class of actions. A simple example of polymorphism is found in the steering wheel of an automobile. The steering wheel (the interface) is the same no matter what type of actual steering mechanism is used. That is, the steering wheel works the same whether your car has manual steering, power steering, or rack-and-pinion steering. Thus, turning the steering wheel left causes the car to go left no matter what type of steering is used. The benefit of the uniform interface is, of course, that once you know how to operate the steering wheel, you can drive any type of car.

The same principle can also apply to programming. For example, consider a *stack* (which is a first-in, last-out list). You might have a program that requires three different types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. In this case, the algorithm that implements each stack is the same, even though the data being stored differs. In a non-object-oriented language, you would be required to create three different sets of stack routines, with each set using different names. However, because of polymorphism, in C# you can create one general set of stack routines that works for all three specific situations. This way, once you know how to use one stack, you can use them all.

More generally, the concept of polymorphism is often expressed by the phrase "one interface, multiple methods." This means that it is possible to design a generic interface to a group of related activities. Polymorphism helps reduce complexity by allowing the same interface to be used to specify a *general class of action.* It is the compiler's job to select the *specific action* (that is, method) as it applies to each situation. You, the programmer, don't need to do this selection manually. You need only remember and utilize the general interface.

Inheritance

Inheritance is the process by which one object can acquire the properties of another object. This is important because it supports the concept of hierarchical classification. If you think about it, most knowledge is made manageable by hierarchical (that is, top-down) classifications. For example, a Red Delicious apple is part of the classification *apple,* which in turn is part of the *fruit* class, which is under the larger class *food.* That is, the *food* class possesses certain qualities (edible, nutritious, and so on) which also, logically, apply to its subclass, *fruit.* In addition to these qualities, the *fruit* class has specific characteristics (juicy, sweet, and so on) that distinguish it from other food. The *apple* class defines those qualities specific to an apple (grows on trees, not tropical, and so on). A Red Delicious apple would, in turn, inherit all the qualities of all preceding classes and would define only those qualities that make it unique.

Without the use of hierarchies, each object would have to explicitly define all of its characteristics. Using inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, the inheritance mechanism makes it possible for one object to be a specific instance of a more general case.

A First Simple Program

It is now time to look at an actual C# program. We will begin by compiling and running the short program shown here:

```
/*
   This is a simple C# program.

   Call this program Example.cs.
*/

using System;

class Example {
```

```
  // A C# program begins with a call to Main().
  static void Main() {
    Console.WriteLine("A simple C# program.");
  }
}
```

The primary development environment for C# is Microsoft's Visual Studio. To compile all of the programs in this book, including those that use the new C# 3.0 features, you will need to use a version of Visual Studio 2008 (or later) that supports C#. A good choice for learning C# 3.0 is Visual C# 2008 Express Edition because (at the time of this writing) it is available free of charge from Microsoft. Visual C# 2008 Express Edition contains a full-featured compiler that supports all of C# 3.0 and is, therefore, able to compile all of the code in this book. It also includes Visual Studio, which is Microsoft's integrated programming environment (IDE). Although the Express Edition does not supply all of the tools that a commercial developer will want, it is perfect for learning C#. At the time of this writing, Visual C# 2008 Express Edition can be downloaded from **microsoft.com/express/**. All of the code in this book has been tested against this compiler.

Using Visual C#, there are two general approaches that you can take to creating, compiling, and running a C# program. First, you can use the Visual Studio IDE. Second, you can use the command-line compiler, **csc.exe**. Both methods are described here.

Using csc.exe, the C# Command-Line Compiler

Although the Visual Studio IDE is what you will probably be using for your commercial projects, some readers will find the C# command-line compiler more convenient, especially for compiling and running the sample programs shown in this book. The reason is that you don't have to create a project for the program. You can simply create the program and then compile it and run it—all from the command line. Therefore, if you know how to use the Command Prompt window and its command-line interface, using the command-line compiler will be faster and easier than using the IDE.

CAUTION *If you are not familiar with the Command Prompt window, then it is probably better to use the Visual Studio IDE. Although its commands are not difficult to learn, trying to learn both the Command Prompt and C# at the same time will be a challenging experience.*

To create and run programs using the C# command-line compiler, follow these three steps:

1. Enter the program using a text editor.
2. Compile the program using **csc.exe**.
3. Run the program.

Entering the Program

The source code for programs shown in this book are available at **www.mhprofessional.com**. However, if you want to enter the programs by hand, you are free to do so. In this case, you must enter the program into your computer using a text editor, such as Notepad. Remember,

you must create text-only files, not formatted word-processor files, because the format information in a word processor file will confuse the C# compiler. When entering the program, call the file **Example.cs**.

Compiling the Program

To compile the program, execute the C# compiler, **csc.exe**, specifying the name of the source file on the command line, as shown here:

```
C:\>csc Example.cs
```

The **csc** compiler creates a file called **Example.exe** that contains the MSIL version of the program. Although MSIL is not executable code, it is still contained in an **exe** file. The Common Language Runtime automatically invokes the JIT compiler when you attempt to execute **Example.exe**. Be aware, however, that if you try to execute **Example.exe** (or any other **exe** file that contains MSIL) on a computer for which the .NET Framework is not installed, the program will not execute because the CLR will be missing.

NOTE *Prior to running* **csc.exe** *you will need to open a Command Prompt window that is configured for Visual Studio. The easiest way to do this is to select Visual Studio 2008 Command Prompt under Visual Studio Tools in the Start menu. Alternatively, you can start an unconfigured Command Prompt window and then run the batch file* **vsvars32.bat**, *which is provided by Visual Studio. You may, however, encounter a problem with the command-line approach. At the time of this writing, Visual C# 2008 Express Edition does not provide the Visual Studio Tools menu or the* **vsvars32.bat** *file. Therefore, if you are using Visual C# 2008 Express, you may not be able to configure a command prompt window automatically. In this case, use the Visual Studio IDE instead. However, Visual C++ 2008 Express Edition does supply both* **vsvars32.bat** *and the Visual Studio 2008 Command Prompt menu selection. Therefore, if you also install Visual C++ 2008 Express Edition, you will be able to start a properly configured command prompt window that will also work for C#.*

Running the Program

To actually run the program, just type its name on the command line, as shown here:

```
C:\>Example
```

When the program is run, the following output is displayed:

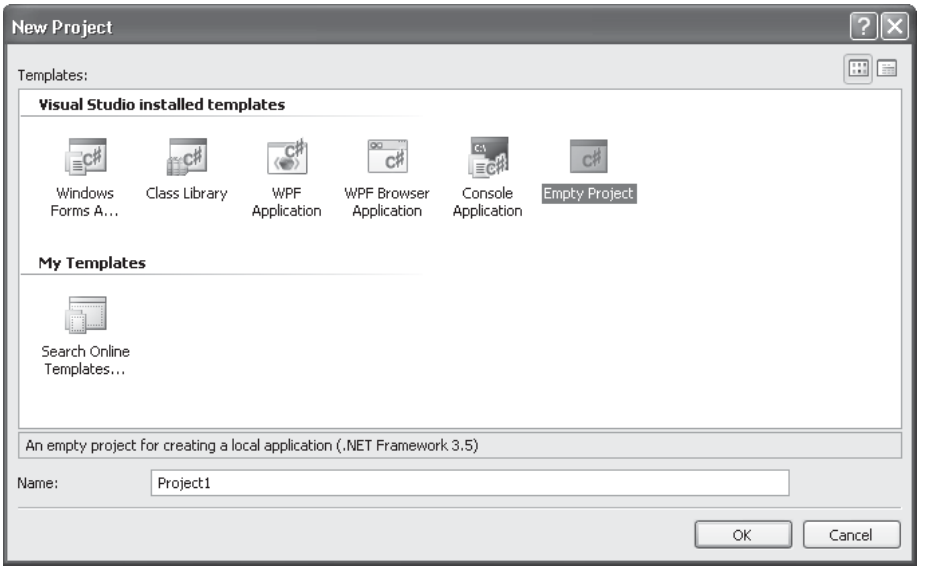
```
A simple C# program.
```

Using the Visual Studio IDE

Visual Studio is Microsoft's integrated programming environment. It lets you edit, compile, run, and debug a C# program, all without leaving its well-thought-out environment. Visual Studio offers convenience and helps manage your programs. It is most effective for larger projects, but it can be used to great success with smaller programs, such as those that constitute the examples in this book.

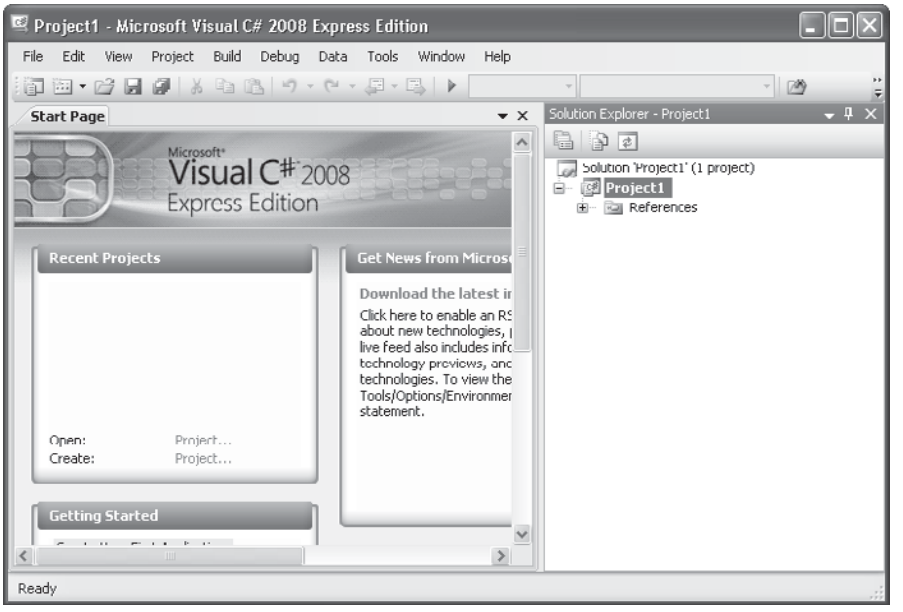
The steps required to edit, compile, and run a C# program using the Visual Studio 2008 IDE are shown here. These steps assume the IDE provided by Visual C# 2008 Express Edition. Slight differences may exist with other versions of Visual Studio 2008.

1. Create a new, empty C# project by selecting File | New Project. Next, select Empty Project:



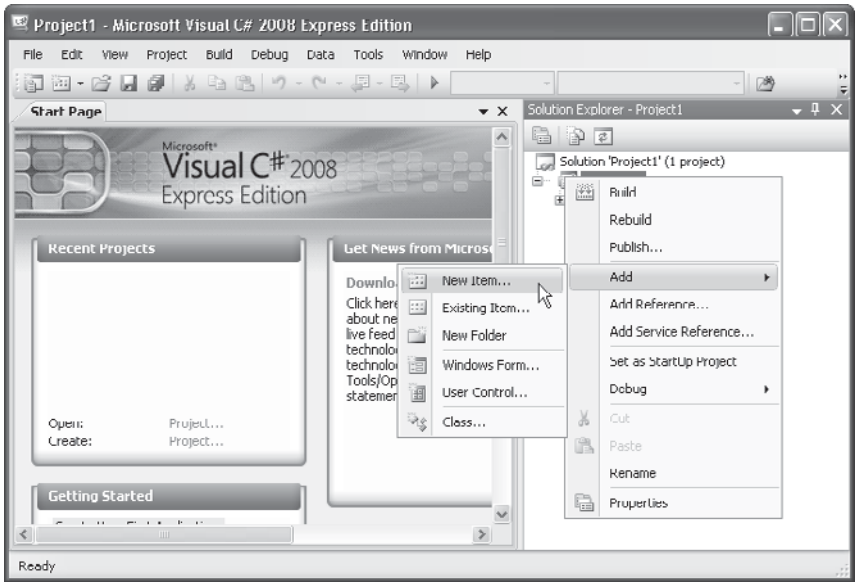
   Then, press OK to create the project.

2. Once the new project is created, the Visual Studio IDE will look like this:

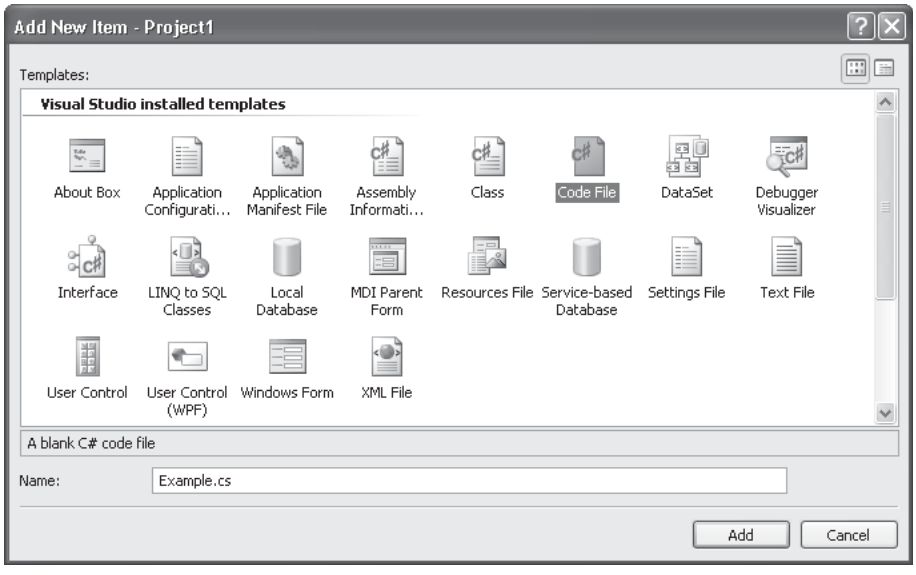


If for some reason you do not see the Solution Explorer window, activate it by selecting Solution Explorer from the View menu.

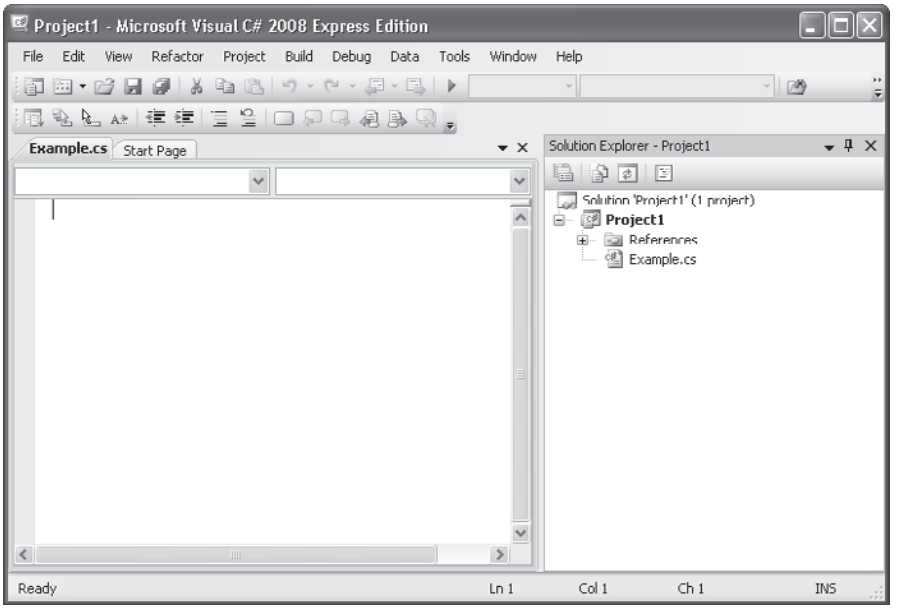
3. At this point, the project is empty and you will need to add a C# source file to it. Do this by right-clicking on Project1 in the Solution Explorer and then selecting Add. You will see the following:



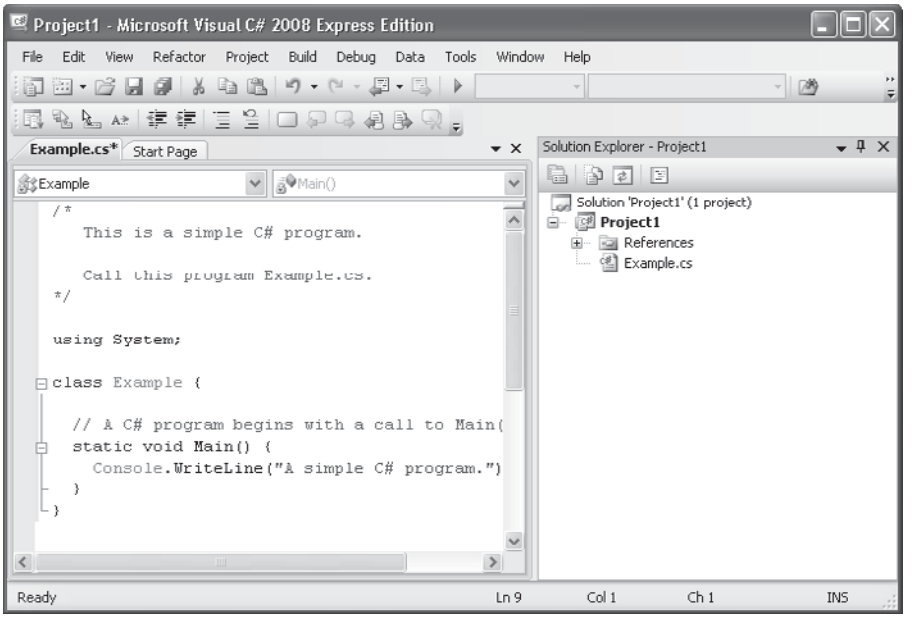
4. Next, select New Item. This causes the Add New Item dialog to be displayed. Select Code File and then change the name to **Example.cs**, as shown here:



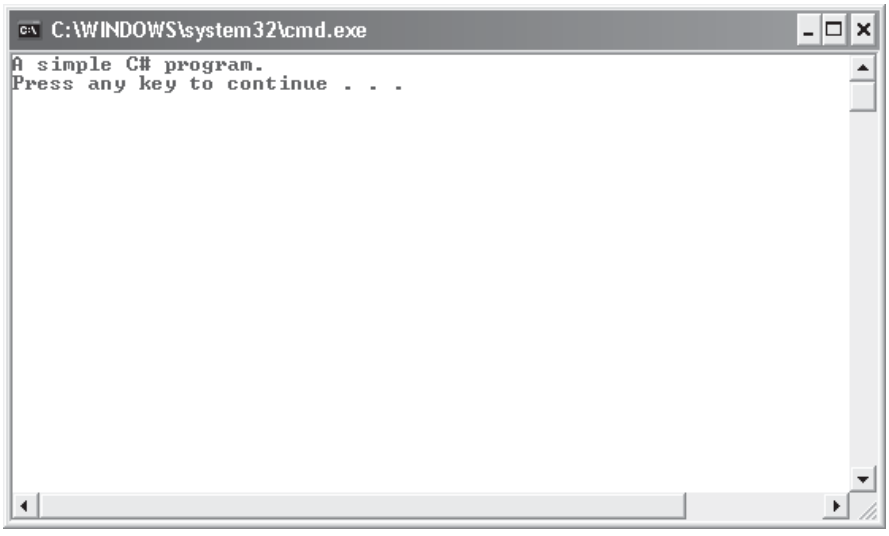
5. Next, add the file to the project by pressing Add. Your screen will now look like this:



6. Next, type the example program into the Example.cs window and then save the file. (You can download the source code to the programs in this book from **www.mhprofessional.com** so you won't have to type in each example manually.) When done, your screen will look like this:



7. Compile the program by selecting Build Solution from the Build menu.
8. Run the program by selecting Start Without Debugging from the Debug menu. When you run the program, you will see the window shown here.



As the preceding instructions show, compiling short sample programs using the IDE involves a number of steps. However, you don't need to create a new project for each example program in this book. Instead, you can use the same C# project. Just delete the current source file and add the new file. Then recompile and run. This approach greatly simplifies the process. Understand, however, that for real-world applications, each program will use its own project.

NOTE *Although the preceding instructions are sufficient to compile and run the programs in this book, if you will be using the Visual Studio IDE for your main work environment, you should become familiar with all of its capabilities and features. It is a very powerful development environment that helps make large projects manageable. The IDE also provides a way of organizing the files and resources associated with a project. It is worth the time and effort that you spend to become proficient at running Visual Studio.*

The First Sample Program, Line by Line

Although **Example.cs** is quite short, it includes several key features that are common to all C# programs. Let's closely examine each part of the program, beginning with its name.

The name of a C# program is arbitrary. Unlike some computer languages (most notably, Java) in which the name of a program file is very important, this is not the case for C#. You were told to call the sample program **Example.cs** so that the instructions for compiling and running the program would apply, but as far as C# is concerned, you could have called the file by another name. For example, the preceding sample program could have been called **Sample.cs**, **Test.cs**, or even **X.cs**.

By convention, C# programs use the **.cs** file extension, and this is a convention that you should follow. Also, many programmers call a file by the name of the principal class defined within the file. This is why the filename **Example.cs** was chosen. Since the names of C# programs are arbitrary, names won't be specified for most of the sample programs in this book. Just use names of your own choosing.

The program begins with the following lines:

```
/*
   This is a simple C# program.

   Call this program Example.cs.
*/
```

This is a *comment.* Like most other programming languages, C# lets you enter a remark into a program's source file. The contents of a comment are ignored by the compiler. Instead, a comment describes or explains the operation of the program to anyone who is reading its source code. In this case, the comment describes the program and reminds you to call the source file **Example.cs**. Of course, in real applications, comments generally explain how some part of the program works or what a specific feature does.

C# supports three styles of comments. The one shown at the top of the program is called a *multiline comment.* This type of comment must begin with **/*** and end with ***/**. Anything between these two comment symbols is ignored by the compiler. As the name suggests, a multiline comment can be several lines long.

The next line in the program is

```
using System;
```

This line indicates that the program is using the **System** namespace. In C#, a *namespace* defines a declarative region. Although we will examine namespaces in detail later in this book, a brief description is useful now. Through the use of namespaces, it is possible to keep one set of names separate from another. In essence, names declared in one namespace will not conflict with names declared in a different namespace. The namespace used by the program is **System**, which is the namespace reserved for items associated with the .NET Framework class library, which is the library used by C#. The **using** keyword simply states that the program is using the names in the given namespace.

The next line of code in the program is shown here:

```
class Example {
```

This line uses the keyword **class** to declare that a new class is being defined. As mentioned, the class is C#'s basic unit of encapsulation. **Example** is the name of the class. The class definition begins with the opening curly brace ({) and ends with the closing curly brace (}). The elements between the two braces are members of the class. For the moment, don't worry too much about the details of a class except to note that in C#, most program activity occurs within one.

The next line in the program is the *single-line comment*, shown here:

```
// A C# program begins with a call to Main().
```

This is the second type of comment supported by C#. A single-line comment begins with a // and ends at the end of the line. Although styles vary, it is not uncommon for programmers to use multiline comments for longer remarks and single-line comments for brief, line-by-line descriptions. (The third type of comment supported by C# aids in the creation of documentation and is described in Appendix A.)

The next line of code is shown here:

```
static void Main() {
```

This line begins the **Main()** method. As mentioned earlier, in C#, a subroutine is called a method. As the comment preceding it suggests, this is the line at which the program will begin executing. All C# applications begin execution by calling **Main()**. The complete meaning of each part of this line cannot be given now, since it involves a detailed understanding of several other C# features. However, since many of the examples in this book will use this line of code, we will take a brief look at it here.

The line begins with the keyword **static**. A method that is modified by **static** can be called before an object of its class has been created. This is necessary because **Main()** is called at program startup. The keyword **void** indicates that **Main()** does not return a value. As you will see, methods can also return values. The empty parentheses that follow **Main** indicate that no information is passed to **Main()**. Although it is possible to pass information into **Main()**, none is passed in this example. The last character on the line is the {. This signals the start of **Main()**'s body. All of the code that comprises a method will occur between the method's opening curly brace and its closing curly brace.

The next line of code is shown here. Notice that it occurs inside **Main()**.

```
Console.WriteLine("A simple C# program.");
```

This line outputs the string "A simple C# program." followed by a new line on the screen. Output is actually accomplished by the built-in method **WriteLine()**. In this case, **WriteLine()** displays the string that is passed to it. Information that is passed to a method is called an *argument*. In addition to strings, **WriteLine()** can be used to display other types of information. The line begins with **Console**, which is the name of a predefined class that supports console I/O. By connecting **Console** with **WriteLine()**, you are telling the compiler that **WriteLine()** is a member of the **Console** class. The fact that C# uses an object to define console output is further evidence of its object-oriented nature.

Notice that the **WriteLine()** statement ends with a semicolon, as does the **using System** statement earlier in the program. In general, statements in C# end with a semicolon. The exception to this rule are *blocks*, which begin with a { and end with a }. This is why those lines in the program don't end with a semicolon. Blocks provide a mechanism for grouping statements and are discussed later in this chapter.

The first } in the program ends **Main()**, and the last } ends the **Example** class definition.

One last point: C# is case-sensitive. Forgetting this can cause serious problems. For example, if you accidentally type **main** instead of **Main**, or **writeline** instead of **WriteLine**, the preceding program will be incorrect. Furthermore, although the C# compiler *will* compile classes that do not contain a **Main()** method, it has no way to execute them. So, had you mistyped **Main**, you would see an error message that states that **Example.exe** does not have an entry point defined.

Handling Syntax Errors

If you are new to programming, it is important to learn how to interpret and respond to errors that may occur when you try to compile a program. Most compilation errors are caused by typing mistakes. As all programmers soon find out, accidentally typing something incorrectly is quite easy. Fortunately, if you type something wrong, the compiler will report a *syntax error* message when it tries to compile your program. This message gives you the line number at which the error is found and a description of the error itself.

Although the syntax errors reported by the compiler are, obviously, helpful, they sometimes can also be misleading. The C# compiler attempts to make sense out of your source code no matter what you have written. For this reason, the error that is reported may not always reflect the actual cause of the problem. In the preceding program, for example, an accidental omission of the opening curly brace after the **Main( )** method generates the following sequence of errors when compiled by the **csc** command-line compiler. (Similar errors are generated when compiling using the IDE.)

```
EX1.CS(12,21): error CS1002: ; expected
EX1.CS(13,22): error CS1519: Invalid token '(' in class, struct, or
interface member declaration
EX1.CS(15,1): error CS1022: Type or namespace definition, or
end-of-file expected
```

Clearly, the first error message is completely wrong, because what is missing is not a semicolon, but a curly brace. The second two messages are equally confusing.

The point of this discussion is that when your program contains a syntax error, don't necessarily take the compiler's messages at face value. They may be misleading. You may need to "second guess" an error message in order to find the problem. Also, look at the last few lines of code immediately preceding the one in which the error was reported. Sometimes an error will not be reported until several lines after the point at which the error really occurred.

A Small Variation

Although all of the programs in this book will use it, the line

```
using System;
```

at the start of the first example program is not technically needed. It is, however, a valuable convenience. The reason it's not necessary is that in C# you can always *fully qualify* a name with the namespace to which it belongs. For example, the line

```
Console.WriteLine("A simple C# program.");
```

can be rewritten as

```
System.Console.WriteLine("A simple C# program.");
```

Thus, the first example could be recoded as shown here:

```
// This version does not include "using System;".

class Example {
```

```
  // A C# program begins with a call to Main().
  static void Main() {

    // Here, Console.WriteLine is fully qualified.
    System.Console.WriteLine("A simple C# program.");
  }
}
```

Since it is quite tedious to always specify the **System** namespace whenever a member of that namespace is used, most C# programmers include **using System** at the top of their programs, as will all of the programs in this book. It is important to understand, however, that you can explicitly qualify a name with its namespace if needed.

A Second Simple Program

Perhaps no other construct is as important to a programming language as the variable. A *variable* is a named memory location that can be assigned a value. It is called a variable because its value can be changed during the execution of a program. In other words, the content of a variable is changeable, not fixed.

The following program creates two variables called **x** and **y**.

```
// This program demonstrates variables.

using System;

class Example2 {
  static void Main() {
    int x; // this declares a variable
    int y; // this declares another variable

    x = 100; // this assigns 100 to x

    Console.WriteLine("x contains " + x);

    y = x / 2;

    Console.Write("y contains x / 2: ");
    Console.WriteLine(y);
  }
}
```

When you run this program, you will see the following output:

```
x contains 100
y contains x / 2: 50
```

This program introduces several new concepts. First, the statement

```
int x; // this declares a variable
```

declares a variable called **x** of type integer. In C#, all variables must be declared before they are used. Further, the kind of values that the variable can hold must also be specified. This is called the *type* of the variable. In this case, **x** can hold integer values. These are whole

numbers. In C#, to declare a variable to be of type integer, precede its name with the keyword **int**. Thus, the preceding statement declares a variable called **x** of type **int**.

The next line declares a second variable called **y**.

```
int y; // this declares another variable
```

Notice that it uses the same format as the first except that the name of the variable is different.

In general, to declare a variable, you will use a statement like this:

type var-name;

Here, *type* specifies the type of variable being declared, and *var-name* is the name of the variable. In addition to **int**, C# supports several other data types.

The following line of code assigns **x** the value **100**:

```
x = 100; // this assigns 100 to x
```

In C#, the assignment operator is the single equal sign. It copies the value on its right side into the variable on its left.

The next line of code outputs the value of **x** preceded by the string "x contains ".

```
Console.WriteLine("x contains " + x);
```

In this statement, the plus sign causes the value of **x** to be displayed after the string that precedes it. This approach can be generalized. Using the **+** operator, you can chain together as many items as you want within a single **WriteLine( )** statement.

The next line of code assigns **y** the value of **x** divided by **2**:

```
y = x / 2;
```

This line divides the value in **x** by 2 and then stores that result in **y**. Thus, after the line executes, **y** will contain the value 50. The value of **x** will be unchanged. Like most other computer languages, C# supports a full range of arithmetic operators, including those shown here:

+	Addition
–	Subtraction
*	Multiplication
/	Division

Here are the next two lines in the program:

```
Console.Write("y contains x / 2: ");
Console.WriteLine(y);
```

Two new things are occurring here. First, the built-in method **Write( )** is used to display the string "y contains x / 2: ". This string is *not* followed by a new line. This means that when the next output is generated, it will start on the same line. The **Write( )** method is just like **WriteLine( )**, except that it does not output a new line after each call. Second, in the call

to **WriteLine()**, notice that **y** is used by itself. Both **Write()** and **WriteLine()** can be used to output values of any of C#'s built-in types.

One more point about declaring variables before we move on: It is possible to declare two or more variables using the same declaration statement. Just separate their names by commas. For example, **x** and **y** could have been declared like this:

```
int x, y; // both declared using one statement
```

NOTE *C# 3.0 includes a new feature called an* implicitly typed variable. *Implicitly typed variables are variables whose type is automatically determined by the compiler. Implicitly typed variables are discussed in Chapter 3.*

Another Data Type

In the preceding program, a variable of type **int** was used. However, an **int** variable can hold only whole numbers. It cannot be used when a fractional component is required. For example, an **int** variable can hold the value 18, but not the value 18.3. Fortunately, **int** is only one of several data types defined by C#. To allow numbers with fractional components, C# defines two floating-point types: **float** and **double**, which represent single- and double-precision values, respectively. Of the two, **double** is the most commonly used.

To declare a variable of type **double**, use a statement similar to that shown here:

```
double result;
```

Here, **result** is the name of the variable, which is of type **double**. Because **result** has a floating-point type, it can hold values such as 122.23, 0.034, or –19.0.

To better understand the difference between **int** and **double**, try the following program:

```
/*
   This program illustrates the differences
   between int and double.
*/

using System;

class Example3 {
  static void Main() {
    int ivar;     // this declares an int variable
    double dvar;  // this declares a floating-point variable

    ivar = 100;   // assign ivar the value 100

    dvar = 100.0; // assign dvar the value 100.0

    Console.WriteLine("Original value of ivar: " + ivar);
    Console.WriteLine("Original value of dvar: " + dvar);

    Console.WriteLine(); // print a blank line

    // Now, divide both by 3.
    ivar = ivar / 3;
    dvar = dvar / 3.0;
```

```
    Console.WriteLine("ivar after division: " + ivar);
    Console.WriteLine("dvar after division: " + dvar);
  }
}
```

The output from this program is shown here:

```
Original value of ivar: 100
Original value of dvar: 100

ivar after division: 33
dvar after division: 33.3333333333333
```

As you can see, when **ivar** (an **int** variable) is divided by 3, a whole-number division is performed, and the outcome is 33—the fractional component is lost. However, when **dvar** (a **double** variable) is divided by 3, the fractional component is preserved.

As the program shows, when you want to specify a floating-point value in a program, you must include a decimal point. If you don't, it will be interpreted as an integer. For example, in C#, the value 100 is an integer, but the value 100.0 is a floating-point value.

There is one other new thing to notice in the program. To print a blank line, simply call **WriteLine()** without any arguments.

The floating-point data types are often used when working with real-world quantities where fractional components are commonly needed. For example, this program computes the area of a circle. It uses the value 3.1416 for pi.

```
// Compute the area of a circle.

using System;

class Circle {
  static void Main() {
    double radius;
    double area;

    radius = 10.0;
    area = radius * radius * 3.1416;

    Console.WriteLine("Area is " + area);
  }
}
```

The output from the program is shown here:

```
Area is 314.16
```

Clearly, the computation of a circle's area could not be satisfactorily achieved without the use of floating-point data.

Two Control Statements

Inside a method, execution proceeds from one statement to the next, top to bottom. It is possible to alter this flow through the use of the various program control statements supported by C#. Although we will look closely at control statements later, two are briefly introduced here because we will be using them to write sample programs.

The if Statement

You can selectively execute part of a program through the use of C#'s conditional statement: the **if**. The **if** statement works in C# much like the IF statement in any other language. For example, it is syntactically identical to the **if** statements in C, C++, and Java. Its simplest form is shown here:

if(*condition*) *statement*;

Here, *condition* is a Boolean (that is, true or false) expression. If *condition* is true, then the statement is executed. If *condition* is false, then the statement is bypassed. Here is an example:

```
if(10 < 11) Console.WriteLine("10 is less than 11");
```

In this case, since 10 is less than 11, the conditional expression is true, and **WriteLine()** will execute. However, consider the following:

```
if(10 < 9) Console.WriteLine("this won't be displayed");
```

In this case, 10 is not less than 9. Thus, the call to **WriteLine()** will not take place.

C# defines a full complement of relational operators that can be used in a conditional expression. They are shown here:

Operator	Meaning
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
= =	Equal to
!=	Not equal

Here is a program that illustrates the **if** statement:

```
// Demonstrate the if.

using System;

class IfDemo {
  static void Main() {
    int a, b, c;

    a = 2;
    b = 3;

    if(a < b) Console.WriteLine("a is less than b");

    // This won't display anything.
    if(a == b) Console.WriteLine("you won't see this");

    Console.WriteLine();
```

```
    c = a - b; // c contains -1

    Console.WriteLine("c contains -1");
    if(c >= 0) Console.WriteLine("c is non-negative");
    if(c < 0) Console.WriteLine("c is negative");

    Console.WriteLine();

    c = b - a; // c now contains 1
    Console.WriteLine("c contains 1");
    if(c >= 0) Console.WriteLine("c is non-negative");
    if(c < 0) Console.WriteLine("c is negative");
  }
}
```

The output generated by this program is shown here:

```
a is less than b

c contains -1
c is negative

c contains 1
c is non-negative
```

Notice one other thing in this program. The line

```
int a, b, c;
```

declares three variables, **a**, **b**, and **c**, by use of a comma-separated list. As mentioned earlier, when you need two or more variables of the same type, they can be declared in one statement. Just separate the variable names with commas.

The for Loop

You can repeatedly execute a sequence of code by creating a *loop.* C# supplies a powerful assortment of loop constructs. The one we will look at here is the **for** loop. Like the **if** statement, the C# **for** loop is similar to its counterpart in C, C++, and Java. The simplest form of the **for** loop is shown here:

for(*initialization*; *condition*; *iteration*) *statement*;

In its most common form, the *initialization* portion of the loop sets a loop control variable to an initial value. The *condition* is a Boolean expression that tests the loop control variable. If the outcome of that test is true, the **for** loop continues to iterate. If it is false, the loop terminates. The *iteration* expression determines how the loop control variable is changed each time the loop iterates. Here is a short program that illustrates the **for** loop:

```
// Demonstrate the for loop.

using System;

class ForDemo {
  static void Main() {
    int count;
```

```
    for(count = 0; count < 5; count = count+1)
      Console.WriteLine("This is count: " + count);

    Console.WriteLine("Done!");
  }
}
```

The output generated by the program is shown here:

```
This is count: 0
This is count: 1
This is count: 2
This is count: 3
This is count: 4
Done!
```

In this example, **count** is the loop control variable. It is set to zero in the initialization portion of the **for**. At the start of each iteration (including the first one), the conditional test **count < 5** is performed. If the outcome of this test is true, the **WriteLine()** statement is executed. Next, the iteration portion of the loop is executed, which adds 1 to **count**. This process continues until **count** reaches 5. At this point, the conditional test becomes false, causing the loop to terminate. Execution picks up at the bottom of the loop.

As a point of interest, in professionally written C# programs you will almost never see the iteration portion of the loop written as shown in the preceding program. That is, you will seldom see statements like this:

```
count = count + 1;
```

The reason is that C# includes a special increment operator that performs this operation. The increment operator is **++** (that is, two consecutive plus signs). The increment operator increases its operand by one. By use of the increment operator, the preceding statement can be written like this:

```
count++;
```

Thus, the **for** in the preceding program will usually be written like this:

```
for(count = 0; count < 5; count++)
```

You might want to try this. As you will see, the loop still runs exactly the same as it did before.

C# also provides a decrement operator, which is specified as **– –**. This operator decreases its operand by one.

Using Code Blocks

Another key element of C# is the *code block.* A code block is a grouping of statements. This is done by enclosing the statements between opening and closing curly braces. Once a block of code has been created, it becomes a logical unit that can be used any place a single statement can. For example, a block can be a target for **if** and **for** statements. Consider this **if** statement:

```
if(w < h) {
  v = w * h;
```

```
  w = 0;
}
```

Here, if **w** is less than **h**, then both statements inside the block will be executed. Thus, the two statements inside the block form a logical unit, and one statement cannot execute without the other also executing. The key point here is that whenever you need to logically link two or more statements, you do so by creating a block. Code blocks allow many algorithms to be implemented with greater clarity and efficiency.

Here is a program that uses a code block to prevent a division by zero:

```
// Demonstrate a block of code.

using System;

class BlockDemo {
  static void Main() {
    int i, j, d;

    i = 5;
    j = 10;

    // The target of this if is a block.
    if(i != 0) {
      Console.WriteLine("i does not equal zero");
      d = j / i;
      Console.WriteLine("j / i is " + d);
    }
  }
}
```

The output generated by this program is shown here:

```
i does not equal zero
j / i is 2
```

In this case, the target of the **if** statement is a block of code and not just a single statement. If the condition controlling the **if** is true (as it is in this case), the three statements inside the block will be executed. Try setting **i** to zero and observe the result.

Here is another example. It uses a code block to compute the sum and the product of the numbers from 1 to 10.

```
// Compute the sum and product of the numbers from 1 to 10.

using System;

class ProdSum {
  static void Main() {
    int prod;
    int sum;
    int i;

    sum = 0;
    prod = 1;
```

```
    for(i=1; i <= 10; i++) {
      sum = sum + i;
      prod = prod * i;
    }
    Console.WriteLine("Sum is " + sum);
    Console.WriteLine("Product is " + prod);

  }
}
```

The output is shown here:

```
Sum is 55
Product is 3628800
```

Here, the block enables one loop to compute both the sum and the product. Without the use of the block, two separate **for** loops would have been required.

One last point: Code blocks do not introduce any runtime inefficiencies. In other words, the { and } do not consume any extra time during the execution of a program. In fact, because of their ability to simplify (and clarify) the coding of certain algorithms, the use of code blocks generally results in increased speed and efficiency.

Semicolons, Positioning, and Indentation

In C#, the semicolon signals the end of a statement. That is, each individual statement must end with a semicolon.

As you know, a block is a set of logically connected statements that are surrounded by opening and closing braces. A block is *not* terminated with a semicolon. Since a block is a group of statements, it makes sense that a block is not terminated by a semicolon; instead, the end of the block is indicated by the closing brace.

C# does not recognize the end of the line as the end of a statement—only a semicolon terminates a statement. For this reason, it does not matter where on a line you put a statement. For example, to C#,

```
x = y;
y = y + 1;
Console.WriteLine(x + " " + y);
```

is the same as

```
x = y;  y = y + 1;  Console.WriteLine(x + " " + y);
```

Furthermore, the individual elements of a statement can also be put on separate lines. For example, the following is perfectly acceptable:

```
Console.WriteLine("This is a long line of output" +
                  x + y + z +
                  "more output");
```

Breaking long lines in this fashion is often used to make programs more readable. It can also help prevent excessively long lines from wrapping.

You may have noticed in the previous examples that certain statements were indented. C# is a free-form language, meaning that it does not matter where you place statements relative to each other on a line. However, over the years, a common and accepted indentation style has developed that allows for very readable programs. This book follows that style, and it is recommended that you do so as well. Using this style, you indent one level after each opening brace and move back out one level after each closing brace. There are certain statements that encourage some additional indenting; these will be covered later.

The C# Keywords

At its foundation, a computer language is defined by its keywords because they determine the features built into the language. C# defines two general types of keywords: *reserved* and *contextual.* The reserved keywords cannot be used as names for variables, classes, or methods. They can be used only as keywords. This is why they are called *reserved.* The terms *reserved words* or *reserved identifiers* are also sometimes used. There are currently 77 reserved keywords defined by version 3.0 of the C# language. They are shown in Table 2-1.

C# 3.0 defines 13 contextual keywords that have a special meaning in certain contexts. In those contexts, they act as keywords. Outside those contexts, they can be used as names for other program elements, such as variable names. Thus, they are not technically reserved. As a general rule, however, you should consider the contextual keywords reserved and avoid using them for any other purpose. Using a contextual keyword as a name for some other program element can be confusing and is considered bad practice by many programmers. The contextual keywords are shown in Table 2-2.

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	volatile
void	while			

Table 2-1 The C# Reserved Keywords

from	get	group	into	join
let	orderby	partial	select	set
value	where	yield		

TABLE 2-2 The C# Contextual Keywords

Identifiers

In C#, an identifier is a name assigned to a method, a variable, or any other user-defined item. Identifiers can be one or more characters long. Variable names may start with any letter of the alphabet or an underscore. Next may be a letter, a digit, or an underscore. The underscore can be used to enhance the readability of a variable name, as in **line_count**. However, identifers containing two consecutive underscores, such as **max_ _value**, are reserved for use by the compiler. Uppercase and lowercase are different; that is, to C#, **myvar** and **MyVar** are separate names. Here are some examples of acceptable identifiers:

Test	x	y2	MaxLoad
up	_top	my_var	sample23

Remember, you can't start an identifier with a digit. Thus, **12x** is invalid, for example. Good programming practice dictates that you choose identifiers that reflect the meaning or usage of the items being named.

Although you cannot use any of the reserved C# keywords as identifiers, C# does allow you to precede a keyword with an @, allowing it to be a legal identifier. For example, **@for** is a valid identifier. In this case, the identifier is actually **for** and the @ is ignored. Here is a program that illustrates the use of an @ identifier:

```
// Demonstrate an @ identifier.

using System;

class IdTest {
  static void Main() {
    int @if; // use if as an identifier

    for(@if = 0; @if < 10; @if++)
      Console.WriteLine("@if is " + @if);
  }
}
```

The output shown here proves the **@if** is properly interpreted as an identifier:

```
@if is 0
@if is 1
@if is 2
@if is 3
@if is 4
```

```
@if is 5
@if is 6
@if is 7
@if is 8
@if is 9
```

Frankly, using @-qualified keywords for identifiers is not recommended, except for special purposes. Also, the @ can precede any identifier, but this is considered bad practice.

The .NET Framework Class Library

The sample programs shown in this chapter make use of two built-in methods: **WriteLine()** and **Write()**. As mentioned, these methods are members of the **Console** class, which is part of the **System** namespace, which is defined by the .NET Framework's class library. As explained earlier in this chapter, the C# environment relies on the .NET Framework class library to provide support for such things as I/O, string handling, networking, and GUIs. Thus, C# as a totality is a combination of the C# language itself, plus the .NET standard classes. As you will see, the class library provides much of the functionality that is part of any C# program. Indeed, part of becoming a C# programmer is learning to use these standard classes. Throughout Part I, various elements of the .NET library classes and methods are described. Part II examines portions of the .NET library in detail.

CHAPTER 3

Data Types, Literals, and Variables

This chapter examines three fundamental elements of C#: data types, literals, and variables. In general, the types of data that a language provides define the kinds of problems to which the language can be applied. As you might expect, C# offers a rich set of built-in data types, which makes C# suitable for a wide range of applications. You can create variables of any of these types, and you can specify constants of each type, which in the language of C# are called *literals.*

Why Data Types Are Important

Data types are especially important in C# because it is a strongly typed language. This means that all operations are type-checked by the compiler for type compatibility. Illegal operations will not be compiled. Thus, strong type-checking helps prevent errors and enhances reliability. To enable strong type-checking, all variables, expressions, and values have a type. There is no concept of a "typeless" variable, for example. Furthermore, a value's type determines what operations are allowed on it. An operation allowed on one type might not be allowed on another.

C#'s Value Types

C# contains two general categories of built-in data types: *value types* and *reference types.* The difference between the two types is what a variable contains. For a value type, a variable holds an actual value, such 3.1416 or 212. For a reference type, a variable holds a reference to the value. The most commonly used reference type is the class, and a discussion of classes and reference types is deferred until later in this book. The value types are described here.

At the core of C# are the 13 value types shown in Table 3-1. Collectively, these are referred to as the *simple types.* They are called simple types because they consist of a single value. (In other words, they are not a composite of two or more values.) They form the foundation of C#'s type system, providing the basic, low-level data elements upon which a program operates. The simple types are also sometimes referred to as *primitive types.*

Type	Meaning
bool	Represents true/false values
byte	8-bit unsigned integer
char	Character
decimal	Numeric type for financial calculations
double	Double-precision floating point
float	Single-precision floating point
int	Integer
long	Long integer
sbyte	8-bit signed integer
short	Short integer
uint	An unsigned integer
ulong	An unsigned long integer
ushort	An unsigned short integer

TABLE 3-1 The C# Value Types

C# strictly specifies a range and behavior for each value type. Because of portability requirements, C# is uncompromising on this account. For example, an **int** is the same in all execution environments. There is no need to rewrite code to fit a specific platform. Although strictly specifying the size of the value types may cause a small loss of performance in some environments, it is necessary in order to achieve portability.

NOTE *In addition to the simple types, C# defines three other categories of value types. These are enumerations, structures, and nullable types, all of which are described later in this book.*

Integers

C# defines nine integer types: **char**, **byte**, **sbyte**, **short**, **ushort**, **int**, **uint**, **long**, and **ulong**. However, the **char** type is primarily used for representing characters, and it is discussed later in this chapter. The remaining eight integer types are used for numeric calculations. Their bit-width and ranges are shown here:

Type	Width in Bits	Range
byte	8	0 to 255
sbyte	8	–128 to 127
short	16	–32,768 to 32,767
ushort	16	0 to 65,535
int	32	–2,147,483,648 to 2,147,483,647
uint	32	0 to 4,294,967,295
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
ulong	64	0 to 18,446,744,073,709,551,615

As the table shows, C# defines both signed and unsigned versions of the various integer types. The difference between signed and unsigned integers is in the way the high-order bit of the integer is interpreted. If a signed integer is specified, then the C# compiler will generate code that assumes the high-order bit of an integer is to be used as a *sign flag*. If the sign flag is 0, then the number is positive; if it is 1, then the number is negative. Negative numbers are almost always represented using the *two's complement* approach. In this method, all bits in the negative number are reversed, and then 1 is added to this number.

Signed integers are important for a great many algorithms, but they have only half the absolute magnitude of their unsigned relatives. For example, as a **short**, here is 32,767:

```
0 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1
```

For a signed value, if the high-order bit were set to 1, the number would then be interpreted as –1 (assuming the two's complement format). However, if you declared this to be a **ushort**, then when the high-order bit was set to 1, the number would become 65,535.

Probably the most commonly used integer type is **int**. Variables of type **int** are often employed to control loops, to index arrays, and for general-purpose integer math. When you need an integer that has a range greater than **int**, you have many options. If the value you want to store is unsigned, you can use **uint**. For large signed values, use **long**. For large unsigned values, use **ulong**. For example, here is a program that computes the distance from the Earth to the sun, in inches. Because this value is so large, the program uses a **long** variable to hold it.

```
// Compute the distance from the Earth to the sun, in inches.

using System;

class Inches {
  static void Main() {
    long inches;
    long miles;

    miles = 93000000; // 93,000,000 miles to the sun

    // 5,280 feet in a mile, 12 inches in a foot.
    inches = miles * 5280 * 12;

    Console.WriteLine("Distance to the sun: " +
                      inches + " inches.");

  }
}
```

Here is the output from the program:

```
Distance to the sun: 5892480000000 inches.
```

Clearly, the result could not have been held in an **int** or **uint** variable.

The smallest integer types are **byte** and **sbyte**. The **byte** type is an unsigned value between 0 and 255. Variables of type **byte** are especially useful when working with raw binary data, such as a byte stream produced by some device. For small signed integers, use **sbyte**. Here is an example that uses a variable of type **byte** to control a **for** loop that produces the summation of the number 100:

```
// Use byte.
```

```
using System;

class Use_byte {
  static void Main() {
    byte x;
    int sum;

    sum = 0;
    for(x = 1; x <= 100; x++)
      sum = sum + x;

    Console.WriteLine("Summation of 100 is " + sum);
  }
}
```

The output from the program is shown here:

```
Summation of 100 is 5050
```

Since the **for** loop runs only from 0 to 100, which is well within the range of a **byte**, there is no need to use a larger type variable to control it.

When you need an integer that is larger than a **byte** or **sbyte**, but smaller than an **int** or **uint**, use **short** or **ushort**.

Floating-Point Types

The floating-point types can represent numbers that have fractional components. There are two kinds of floating-point types, **float** and **double**, which represent single- and double-precision numbers, respectively. The type **float** is 32 bits wide and has an approximate range of 1.5E–45 to 3.4E+38. The **double** type is 64 bits wide and has an approximate range of 5E–324 to 1.7E+308.

Of the two, **double** is the most commonly used. One reason for this is that many of the math functions in C#'s class library (which is the .NET Framework library) use **double** values. For example, the **Sqrt( )** method (which is defined by the library class **System.Math**) returns a **double** value that is the square root of its **double** argument. Here, **Sqrt( )** is used to compute the radius of a circle given the circle's area:

```
// Find the radius of a circle given its area.

using System;

class FindRadius {
  static void Main() {
    Double r;
    Double area;

    area = 10.0;

    r = Math.Sqrt(area / 3.1416);
```

```
    Console.WriteLine("Radius is " + r);
  }
}
```

The output from the program is shown here:

```
Radius is 1.78412203012729
```

One other point about the preceding example. As mentioned, **Sqrt( )** is a member of the **Math** class. Notice how **Sqrt( )** is called; it is preceded by the name **Math**. This is similar to the way **Console** precedes **WriteLine( )**. Although not all standard methods are called by specifying their class name first, several are, as the next example shows.

The following program demonstrates several of C#'s trigonometric functions, which are also part of C#'s math library. They also operate on **double** data. The program displays the sine, cosine, and tangent for the angles (measured in radians) from 0.1 to 1.0.

```
//  Demonstrate Math.Sin(), Math.Cos(), and Math.Tan().

using System;

class Trigonometry {
  static void Main() {
    Double theta; // angle in radians

    for(theta = 0.1; theta <= 1.0; theta = theta + 0.1) {
      Console.WriteLine("Sine of " + theta + "  is " +
                        Math.Sin(theta));
      Console.WriteLine("Cosine of " + theta + "  is " +
                        Math.Cos(theta));
      Console.WriteLine("Tangent of " + theta + "  is " +
                        Math.Tan(theta));
      Console.WriteLine();
    }
  }
}
```

Here is a portion of the program's output:

```
Sine of 0.1  is 0.0998334166468282
Cosine of 0.1  is 0.995004165278026
Tangent of 0.1  is 0.100334672085451

Sine of 0.2  is 0.198669330795061
Cosine of 0.2  is 0.980066577841242
Tangent of 0.2  is 0.202710035508673

Sine of 0.3  is 0.29552020666134
Cosine of 0.3  is 0.955336489125606
Tangent of 0.3  is 0.309336249609623
```

To compute the sine, cosine, and tangent, the standard library methods **Math.Sin( )**, **Math.Cos( )**, and **Math.Tan( )** are used. Like **Math.Sqrt( )**, the trigonometric methods are called with a **double** argument, and they return a **double** result. The angles must be specified in radians.

The decimal Type

Perhaps the most interesting C# numeric type is **decimal**, which is intended for use in monetary calculations. The **decimal** type utilizes 128 bits to represent values within the range 1E–28 to 7.9E+28. As you may know, normal floating-point arithmetic is subject to a variety of rounding errors when it is applied to decimal values. The **decimal** type eliminates these errors and can accurately represent up to 28 decimal places (or 29 places in some cases). This ability to represent decimal values without rounding errors makes it especially useful for computations that involve money.

Here is a program that uses a **decimal** type in a financial calculation. The program computes the discounted price given the original price and a discount percentage.

```
// Use the decimal type to compute a discount.

using System;

class UseDecimal {
  static void Main() {
    decimal price;
    decimal discount;
    decimal discounted_price;

    // Compute discounted price.
    price = 19.95m;
    discount = 0.15m; // discount rate is 15%

    discounted_price = price - ( price * discount);

    Console.WriteLine("Discounted price: $" + discounted_price);
  }
}
```

The output from this program is shown here:

```
Discounted price: $16.9575
```

In the program, notice that the decimal constants are followed by the *m* suffix. This is necessary because without the suffix, these values would be interpreted as standard floating-point constants, which are not compatible with the **decimal** data type. You can assign an integer value, such as 10, to a **decimal** variable without the use of the *m* suffix, though. (A detailed discussion of numeric constants is found later in this chapter.)

Here is another example that uses the **decimal** type. It computes the future value of an investment that has a fixed rate of return over a period of years.

```
/*
   Use the decimal type to compute the future value
   of an investment.
*/

using System;

class FutVal {
```

```
  static void Main() {
    decimal amount;
    decimal rate_of_return;
    int years, i;

    amount = 1000.0M;
    rate_of_return = 0.07M;
    years = 10;

    Console.WriteLine("Original investment: $" + amount);
    Console.WriteLine("Rate of return: " + rate_of_return);
    Console.WriteLine("Over " + years + " years");

    for(i = 0; i < years; i++)
      amount = amount + (amount * rate_of_return);

    Console.WriteLine("Future value is $" + amount);
  }
}
```

Here is the output:

```
Original investment: $1000
Rate of return: 0.07
Over 10 years
Future value is $1967.151357289565322490000
```

Notice that the result is accurate to several decimal places—more than you would probably want! Later in this chapter you will see how to format such output in a more appealing fashion.

Characters

In C#, characters are not 8-bit quantities like they are in many other computer languages, such as C++. Instead, C# uses a 16-bit character type called *Unicode*. Unicode defines a character set that is large enough to represent all of the characters found in all human languages. Although many languages, such as English, French, or German, use relatively small alphabets, some languages, such as Chinese, use very large character sets that cannot be represented using just 8 bits. To address this situation, in C#, **char** is an unsigned 16-bit type having a range of 0 to 65,535. The standard 8-bit ASCII character set is a subset of Unicode and ranges from 0 to 127. Thus, the ASCII characters are still valid C# characters.

A character variable can be assigned a value by enclosing the character inside single quotes. For example, this assigns X to the variable **ch**:

```
char ch;
ch = 'X';
```

You can output a **char** value using a **WriteLine( )** statement. For example, this line outputs the value in **ch**:

```
Console.WriteLine("This is ch: " + ch);
```

Although **char** is defined by C# as an integer type, it cannot be freely mixed with integers in all cases. This is because there are no automatic type conversions from integer to **char**. For example, the following fragment is invalid:

```
char ch;

ch = 88; // error, won't work
```

The reason the preceding code will not work is that 10 is an integer value, and it won't automatically convert to a **char**. If you attempt to compile this code, you will see an error message. To make the assignment legal, you would need to employ a cast, which is described later in this chapter.

The bool Type

The **bool** type represents true/false values. C# defines the values true and false using the reserved words **true** and **false**. Thus, a variable or expression of type **bool** will be one of these two values. Furthermore, there is no conversion defined between **bool** and integer values. For example, 1 does not convert to true, and 0 does not convert to false.

Here is a program that demonstrates the **bool** type:

```
// Demonstrate bool values.

using System;

class BoolDemo {
  static void Main() {
    bool b;

    b = false;
    Console.WriteLine("b is " + b);
    b = true;
    Console.WriteLine("b is " + b);

    // A bool value can control the if statement.
    if(b) Console.WriteLine("This is executed.");

    b = false;
    if(b) Console.WriteLine("This is not executed.");

    // Outcome of a relational operator is a bool value.
    Console.WriteLine("10 > 9 is " + (10 > 9));
  }
}
```

The output generated by this program is shown here:

```
b is False
b is True
This is executed.
10 > 9 is True
```

There are three interesting things to notice about this program. First, as you can see, when a **bool** value is output by **WriteLine( )**, "True" or "False" is displayed. Second, the value of a **bool** variable is sufficient, by itself, to control the **if** statement. There is no need to write an **if** statement like this:

```
if(b == true) ...
```

Third, the outcome of a relational operator, such as **<**, is a **bool** value. This is why the expression **10 > 9** displays the value "True." Further, the extra set of parentheses around **10 > 9** is necessary because the **+** operator has a higher precedence than the **>**.

Some Output Options

Up to this point, when data has been output using a **WriteLine( )** statement, it has been displayed using the default format. However, the .NET Framework defines a sophisticated formatting mechanism that gives you detailed control over how data is displayed. Although formatted I/O is covered in detail later in this book, it is useful to introduce some formatting options at this time. Using these options, you will be able to specify the way values look when output via a **WriteLine( )** statement. Doing so enables you to produce more appealing output. Keep in mind that the formatting mechanism supports many more features than described here.

When outputting lists of data, you have been separating each part of the list with a plus sign, as shown here:

```
Console.WriteLine("You ordered " + 2 + " items at $" + 3 + " each.");
```

While very convenient, outputting numeric information in this way does not give you any control over how that information appears. For example, for a floating-point value, you can't control the number of decimal places displayed. Consider the following statement:

```
Console.WriteLine("Here is 10/3: " + 10.0/3.0);
```

It generates this output:

```
Here is 10/3: 3.33333333333333
```

Although this might be fine for some purposes, displaying so many decimal places could be inappropriate for others. For example, in financial calculations, you will usually want to display two decimal places.

To control how numeric data is formatted, you will need to use a second form of **WriteLine( )**, shown here, which allows you to embed formatting information:

WriteLine("*format string*", *arg0*, *arg1*, ... , *argN*);

In this version, the arguments to **WriteLine( )** are separated by commas and not + signs. The *format string* contains two items: regular, printing characters that are displayed as-is, and format specifiers. Format specifiers take this general form:

{*argnum*, *width*: *fmt*}

Here, *argnum* specifies the number of the argument (starting from zero) to display. The minimum width of the field is specified by *width,* and the format is specified by *fmt.* The *width* and *fmt* are optional.

During execution, when a format specifier is encountered in the format string, the corresponding argument, as specified by *argnum,* is substituted and displayed. Thus, the position of a format specification within the format string determines where its matching data will be displayed. Both *width* and *fmt* are optional. Therefore, in its simplest form, a format specifier simply indicates which argument to display. For example, **{0}** indicates *arg0,* **{1}** specifies *arg1,* and so on.

Let's begin with a simple example. The statement

```
Console.WriteLine("February has {0} or {1} days.", 28, 29);
```

produces the following output:

```
February has 28 or 29 days.
```

As you can see, the value 28 is substituted for **{0}**, and 29 is substituted for **{1}**. Thus, the format specifiers identify the location at which the subsequent arguments, in this case 28 and 29, are displayed within the string. Furthermore, notice that the additional values are separated by commas, not + signs.

Here is a variation of the preceding statement that specifies minimum field widths:

```
Console.WriteLine("February has {0,10} or {1,5} days.", 28, 29);
```

It produces the following output:

```
February has         28 or    29 days.
```

As you can see, spaces have been added to fill out the unused portions of the fields. Remember, a minimum field width is just that: the *minimum* width. Output can exceed that width if needed.

Of course, the arguments associated with a format command need not be constants. For example, this program displays a table of squares and cubes. It uses format commands to output the values.

```
// Use format commands.

using System;

class DisplayOptions {
  static void Main() {
    int i;

    Console.WriteLine("Value\tSquared\tCubed");

    for(i = 1; i < 10; i++)
      Console.WriteLine("{0}\t{1}\t{2}", i, i*i, i*i*i);
  }
}
```

The output is shown here:

```
Value   Squared Cubed
1       1       1
2       4       8
3       9       27
4       16      64
5       25      125
6       36      216
7       49      343
8       64      512
9       81      729
```

In the preceding examples, no formatting was applied to the values themselves. Of course, the purpose of using format specifiers is to control the way the data looks. The types of data most commonly formatted are floating-point and decimal values. One of the easiest ways to specify a format is to describe a template that **WriteLine( )** will use. To do this, show an example of the format that you want, using **#**s to mark the digit positions. You can also specify the decimal point and commas. For example, here is a better way to display 10 divided by 3:

```
Console.WriteLine("Here is 10/3: {0:#.##}", 10.0/3.0);
```

The output from this statement is shown here:

```
Here is 10/3: 3.33
```

In this example, the template is **#.##**, which tells **WriteLine( )** to display two decimal places. It is important to understand, however, that **WriteLine( )** will display more than one digit to the left of the decimal point, if necessary, so as not to misrepresent the value.

Here is another example. This statement

```
Console.WriteLine("{0:###,###.##}", 123456.56);
```

generates this output:

```
123,456.56
```

If you want to display monetary values, use the **C** format specifier. For example:

```
decimal balance;

balance = 12323.09m;
Console.WriteLine("Current balance is {0:C}", balance);
```

The output from this sequence is shown here (in U.S. dollar format):

```
Current balance is $12,323.09
```

The **C** format can be used to improve the output from the price discount program shown earlier:

```
// Use the C format specifier to output dollars and cents.

using System;
```

```
class UseDecimal {
  static void Main() {
    decimal price;
    decimal discount;
    decimal discounted_price;

    // Compute discounted price.
    price = 19.95m;
    discount = 0.15m; // discount rate is 15%

    discounted_price = price - ( price * discount);

    Console.WriteLine("Discounted price: {0:C}", discounted_price);
  }
}
```

Here is the way the output now looks:

```
Discounted price: $16.96
```

Literals

In C#, *literals* refer to fixed values that are represented in their human-readable form. For example, the number 100 is a literal. For the most part, literals and their usage are so intuitive that they have been used in one form or another by all the preceding sample programs. Now the time has come to explain them formally.

C# literals can be of any simple type. The way each literal is represented depends upon its type. As explained earlier, character literals are enclosed between single quotes. For example, 'a' and '%' are both character literals.

Integer literals are specified as numbers without fractional components. For example, 10 and –100 are integer literals. Floating-point literals require the use of the decimal point followed by the number's fractional component. For example, 11.123 is a floating-point literal. C# also allows you to use scientific notation for floating-point numbers.

Since C# is a strongly typed language, literals, too, have a type. Naturally, this raises the following question: What is the type of a numeric literal? For example, what is the type of 12, 123987, or 0.23? Fortunately, C# specifies some easy-to-follow rules that answer these questions.

First, for integer literals, the type of the literal is the smallest integer type that will hold it, beginning with **int**. Thus, an integer literal is either of type **int**, **uint**, **long**, or **ulong**, depending upon its value. Second, floating-point literals are of type **double**.

If C#'s default type is not what you want for a literal, you can explicitly specify its type by including a suffix. To specify a **long** literal, append an *l* or an *L*. For example, 12 is an **int**, but 12L is a **long**. To specify an unsigned integer value, append a *u* or *U*. Thus, 100 is an **int**, but 100U is a **uint**. To specify an unsigned, long integer, use *ul* or *UL*. For example, 984375UL is of type **ulong**.

To specify a **float** literal, append an *F* or *f* to the constant. For example, 10.19F is of type **float**. Although redundant, you can specify a **double** literal by appending a *D* or *d*. (As just mentioned, floating-point literals are **double** by default.)

To specify a **decimal** literal, follow its value with an *m* or *M*. For example, 9.95M is a **decimal** literal.

Although integer literals create an **int**, **uint**, **long**, or **ulong** value by default, they can still be assigned to variables of type **byte**, **sbyte**, **short**, or **ushort** as long as the value being assigned can be represented by the target type.

Hexadecimal Literals

As you probably know, in programming it is sometimes easier to use a number system based on 16 instead of 10. The base 16 number system is called *hexadecimal* and uses the digits 0 through 9 plus the letters A through F, which stand for 10, 11, 12, 13, 14, and 15. For example, the hexadecimal number 10 is 16 in decimal. Because of the frequency with which hexadecimal numbers are used, C# allows you to specify integer literals in hexadecimal format. A hexadecimal literal must begin with **0x** (a 0 followed by an *x*). Here are some examples:

```
count = 0xFF; // 255 in decimal
incr = 0x1a;  // 26 in decimal
```

Character Escape Sequences

Enclosing character literals in single quotes works for most printing characters, but a few characters, such as the carriage return, pose a special problem when a text editor is used. In addition, certain other characters, such as the single and double quotes, have special meaning in C#, so you cannot use them directly. For these reasons, C# provides special *escape sequences,* sometimes referred to as *backslash character constants,* shown in Table 3-2. These sequences are used in place of the characters they represent.

For example, this assigns **ch** the tab character:

```
ch = '\t';
```

The next example assigns a single quote to **ch**:

```
ch = '\'';
```

Escape Sequence	Description
\a	Alert (bell)
\b	Backspace
\f	Form feed
\n	New line (linefeed)
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\0	Null
\'	Single quote
\"	Double quote
\\	Backslash

Table 3-2 Character Escape Sequences

String Literals

C# supports one other type of literal: the *string*. A string literal is a set of characters enclosed by double quotes. For example,

```
"this is a test"
```

is a string. You have seen examples of strings in many of the **WriteLine( )** statements in the preceding sample programs.

In addition to normal characters, a string literal can also contain one or more of the escape sequences just described. For example, consider the following program. It uses the **\n** and **\t** escape sequences.

```
// Demonstrate escape sequences in strings.

using System;

class StrDemo {
  static void Main() {
    Console.WriteLine("Line One\nLine Two\nLine Three");
    Console.WriteLine("One\tTwo\tThree");
    Console.WriteLine("Four\tFive\tSix");

    // Embed quotes.
    Console.WriteLine("\"Why?\", he asked.");
  }
}
```

The output is shown here:

```
Line One
Line Two
Line Three
One     Two     Three
Four    Five    Six
"Why?", he asked.
```

Notice how the **\n** escape sequence is used to generate a new line. You don't need to use multiple **WriteLine( )** statements to get multiline output. Just embed **\n** within a longer string at the points where you want the new lines to occur. Also note how a quotation mark is generated inside a string.

In addition to the form of string literal just described, you can also specify a *verbatim string literal*. A verbatim string literal begins with an **@**, which is followed by a quoted string. The contents of the quoted string are accepted without modification and can span two or more lines. Thus, you can include newlines, tabs, and so on, but you don't need to use the escape sequences. The only exception is that to obtain a double quote (“), you must use two double quotes in a row (“”). Here is a program that demonstrates verbatim string literals:

```
// Demonstrate verbatim literal strings.

using System;

class Verbatim {
```

```
  static void Main() {
    Console.WriteLine(@"This is a verbatim
string literal
that spans several lines.
");
    Console.WriteLine(@"Here is some tabbed output:
1       2       3       4
5       6       7       8
");
    Console.WriteLine(@"Programmers say, ""I like C#.""");
  }
}
```

The output from this program is shown here:

```
This is a verbatim
string literal
that spans several lines.

Here is some tabbed output:
1       2       3       4
5       6       7       8

Programmers say, "I like C#."
```

The important point to notice about the preceding program is that the verbatim string literals are displayed precisely as they are entered into the program.

The advantage of verbatim string literals is that you can specify output in your program exactly as it will appear on the screen. However, in the case of multiline strings, the wrapping will obscure the indentation of your program. For this reason, the programs in this book will make only limited use of verbatim string literals. That said, they are still a wonderful benefit for many formatting situations.

One last point: Don't confuse strings with characters. A character literal, such as 'X', represents a single letter of type **char**. A string containing only one letter, such as "X", is still a string.

A Closer Look at Variables

Variables are declared using this form of statement:

type var-name;

where *type* is the data type of the variable and *var-name* is its name. You can declare a variable of any valid type, including the value types just described. It is important to understand that a variable's capabilities are determined by its type. For example, a variable of type **bool** cannot be used to store floating-point values. Furthermore, the type of a variable cannot change during its lifetime. An **int** variable cannot turn into a **char** variable, for example.

All variables in C# must be declared prior to their use. This is necessary because the compiler must know what type of data a variable contains before it can properly compile any statement that uses the variable. It also enables C# to perform strict type-checking.

C# defines several different kinds of variables. The kind that we have been using are called *local variables* because they are declared within a method.

Initializing a Variable

One way to give a variable a value is through an assignment statement, as you have already seen. Another way is by giving it an initial value when it is declared. To do this, follow the variable's name with an equal sign and the value being assigned. The general form of initialization is shown here:

type var-name = value;

Here, *value* is the value that is given to the variable when it is created. The value must be compatible with the specified type.

Here are some examples:

```
int count = 10; // give count an initial value of 10
char ch = 'X';  // initialize ch with the letter X
float f = 1.2F; // f is initialized with 1.2
```

When declaring two or more variables of the same type using a comma-separated list, you can give one or more of those variables an initial value. For example:

```
int a, b = 8, c = 19, d; // b and c have initializations
```

In this case, only **b** and **c** are initialized.

Dynamic Initialization

Although the preceding examples have used only constants as initializers, C# allows variables to be initialized dynamically, using any expression valid at the point at which the variable is declared. For example, here is a short program that computes the hypotenuse of a right triangle given the lengths of its two opposing sides.

```
// Demonstrate dynamic initialization.

using System;

class DynInit {
  static void Main() {
    // Length of sides.
    double s1 = 4.0;
    double s2 = 5.0;

    // Dynamically initialize hypot.
    double hypot = Math.Sqrt( (s1 * s1) + (s2 * s2) );

    Console.Write("Hypotenuse of triangle with sides " +
                  s1 + " by " + s2 + " is ");

    Console.WriteLine("{0:#.###}.", hypot);

  }
}
```

Here is the output:

```
Hypotenuse of triangle with sides 4 by 5 is 6.403.
```

Here, three local variables—**s1**, **s2**, and **hypot**—are declared. The first two, **s1** and **s2**, are initialized by constants. However, **hypot** is initialized dynamically to the length of the hypotenuse. Notice that the initialization involves calling **Math.Sqrt( )**. As explained, you can use any expression that is valid at the point of the initialization. Since a call to **Math.Sqrt( )** (or any other library method) is valid at this point, it can be used in the initialization of **hypot**. The key point here is that the initialization expression can use any element valid at the time of the initialization, including calls to methods, other variables, or literals.

Implicitly Typed Variables

As explained, in C# all variables must be declared. Normally, a declaration includes the type of the variable, such as **int** or **bool**, followed by the name of the variable. However, beginning with C# 3.0, it is possible to let the compiler determine the type of a local variable based on the value used to initialize it. This is called an *implicitly typed variable.*

An implicitly typed variable is declared using the keyword **var**, and it must be initialized. The compiler uses the type of the initializer to determine the type of the variable. Here is an example:

```
var e = 2.7183;
```

Because **e** is initialized with a floating-point literal (whose type is **double** by default), the type of **e** is **double**. Had **e** been declared like this:

```
var e = 2.7183F;
```

then **e** would have the type **float**, instead.

The following program demonstrates implicitly typed variables. It reworks the program shown in the preceding section so that all variables are implicitly typed.

```
//  Demonstrate implicitly typed variables.

using System;

class ImplicitlyTypedVar {
  static void Main() {

    // These are now implicitly typed variables. They
    // are of type double because their initializing
    // expressions are of type double.
    var s1 = 4.0;
    var s2 = 5.0;

    // Now, hypot is implicitly typed.  Its type is double
    // because the return type of Sqrt() is double.
    var hypot = Math.Sqrt( (s1 * s1) + (s2 * s2) );

    Console.Write("Hypotenuse of triangle with sides " +
                  s1 + " by " + s2 + " is ");

    Console.WriteLine("{0:#.###}.", hypot);

    // The following statement will not compile because
    // s1 is a double and cannot be assigned a decimal value.
```

```
//    s1 = 12.2M;  // Error!
  }
}
```

The output is the same as before.

It is important to emphasize that an implicitly typed variable is still a strongly typed variable. Notice this commented-out line in the program:

```
//    s1 = 12.2M;  // Error!
```

This assignment is invalid because **s1** is of type **double**. Thus, it cannot be assigned a **decimal** value. The only difference between an implicitly typed variable and a "normal" explicitly typed variable is how the type is determined. Once that type has been determined, the variable has a type, and this type is fixed throughout the lifetime of the variable. Thus, the type of **s1** cannot be changed during execution of the program.

Implicitly typed variables were not added to C# to replace "normal" variable declarations. Instead, implicitly typed variables are designed to handle some special-case situations, the most important of which relate to Language-Integrated Query (LINQ), which is described in Chapter 19. Therefore, for most variable declarations, you should continue to use explicitly typed variables because they make your code easier to read and easier to understand.

One last point: Only one implicitly typed variable can be declared at any one time. Therefore, the following declaration,

```
var s1 = 4.0, s2 = 5.0; // Error!
```

is wrong and won't compile because it attempts to declare both **s1** and **s2** at the same time.

The Scope and Lifetime of Variables

So far, all of the variables that we have been using are declared at the start of the **Main()** method. However, C# allows a local variable to be declared within any block. As explained in Chapter 1, a block begins with an opening curly brace and ends with a closing curly brace. A block defines a *scope.* Thus, each time you start a new block, you are creating a new scope. A scope determines what names are visible to other parts of your program without qualification. It also determines the lifetime of local variables.

The most important scopes in C# are those defined by a class and those defined by a method. A discussion of class scope (and variables declared within it) is deferred until later in this book, when classes are described. For now, we will examine only the scopes defined by or within a method.

The scope defined by a method begins with its opening curly brace and ends with its closing curly brace. However, if that method has parameters, they too are included within the scope defined by the method.

As a general rule, local variables declared inside a scope are not visible to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are protecting it from access or modification from outside the scope. Indeed, the scope rules provide the foundation for encapsulation.

Scopes can be nested. For example, each time you create a block of code, you are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope. This means that local variables declared in the outer scope will be visible to code within the inner scope.

However, the reverse is not true. Local variables declared within the inner scope will not be visible outside it.

To understand the effect of nested scopes, consider the following program:

```
// Demonstrate block scope.

using System;

class ScopeDemo {
  static void Main() {
    int x; // known to all code within Main()

    x = 10;
    if(x == 10) { // start new scope
      int y = 20; // known only to this block

      // x and y both known here.
      Console.WriteLine("x and y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100; // Error! y not known here.

    // x is still known here.
    Console.WriteLine("x is " + x);
  }
}
```

As the comments indicate, the variable **x** is declared at the start of **Main( )**'s scope and is accessible to all subsequent code within **Main( )**. Within the **if** block, **y** is declared. Since a block defines a scope, **y** is visible only to other code within its block. This is why outside of its block, the line **y = 100;** is commented out. If you remove the leading comment symbol, a compile-time error will occur because **y** is not visible outside of its block. Within the **if** block, **x** can be used because code within a block (that is, a nested scope) has access to variables declared by an enclosing scope.

Within a block, variables can be declared at any point, but are valid only after they are declared. Thus, if you define a variable at the start of a method, it is available to all of the code within that method. Conversely, if you declare a variable at the end of a block, it is effectively useless, because no code will have access to it.

If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered. For example, consider this program:

```
// Demonstrate lifetime of a variable.

using System;

class VarInitDemo {
  static void Main() {
    int x;

    for(x = 0; x < 3; x++) {
      int y = -1; // y is initialized each time block is entered
      Console.WriteLine("y is: " + y); // this always prints -1
```

```
      y = 100;
      Console.WriteLine("y is now: " + y);
    }
  }
}
```

The output generated by this program is shown here:

```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

As you can see, **y** is always reinitialized to –1 each time the inner **for** loop is entered. Even though it is subsequently assigned the value 100, this value is lost.

There is one quirk to C#'s scope rules that may surprise you: Although blocks can be nested, no variable declared within an inner scope can have the same name as a variable declared by an enclosing scope. For example, the following program, which tries to declare two separate variables with the same name, will not compile.

```
/*
   This program attempts to declare a variable
   in an inner scope with the same name as one
   defined in an outer scope.

   *** This program will not compile. ***
*/

using System;

class NestVar {
  static void Main() {
    int count;

    for(count = 0; count < 10; count = count+1) {
      Console.WriteLine("This is count: " + count);

      int count; // illegal!!!
      for(count = 0; count < 2; count++)
        Console.WriteLine("This program is in error!");
    }
  }
}
```

If you come from a C/C++ background, then you know that there is no restriction on the names you give variables declared in an inner scope. Thus, in C/C++ the declaration of **count** within the block of the outer **for** loop is completely valid. However, in C/C++, such a declaration hides the outer variable. The designers of C# felt that this type of *name hiding* could easily lead to programming errors and disallowed it.

Type Conversion and Casting

In programming, it is common to assign one type of variable to another. For example, you might want to assign an **int** value to a **float** variable, as shown here:

```
int i;
float f;

i = 10;
f = i; // assign an int to a float
```

When compatible types are mixed in an assignment, the value of the right side is automatically converted to the type of the left side. Thus, in the preceding fragment, the value in **i** is converted into a **float** and then assigned to **f**. However, because of C#'s strict type-checking, not all types are compatible, and thus, not all type conversions are implicitly allowed. For example, **bool** and **int** are not compatible. Fortunately, it is still possible to obtain a conversion between incompatible types by using a *cast*. A cast performs an explicit type conversion. Both automatic type conversion and casting are examined here.

Automatic Conversions

When one type of data is assigned to another type of variable, an *implicit* type conversion will take place automatically if

- The two types are compatible.
- The destination type has a range that is greater than the source type.

When these two conditions are met, a *widening conversion* takes place. For example, the **int** type is always large enough to hold all valid **byte** values, and both **int** and **byte** are compatible integer types, so an implicit conversion can be applied.

For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. For example, the following program is perfectly valid since **long** to **double** is a widening conversion that is automatically performed.

```
// Demonstrate implicit conversion from long to double.

using System;

class LtoD {
  static void Main() {
    long L;
    double D;

    L = 100123285L;
    D = L;

    Console.WriteLine("L and D: " + L + " " + D);
  }
}
```

Although there is an implicit conversion from **long** to **double**, there is no implicit conversion from **double** to **long** since this is not a widening conversion. Thus, the following version of the preceding program is invalid:

```
// *** This program will not compile. ***

using System;

class LtoD {
  static void Main() {
    long L;
    double D;

    D = 100123285.0;
    L = D; // Illegal!!!

    Console.WriteLine("L and D: " + L + " " + D);

  }
}
```

In addition to the restrictions just described, there are no implicit conversions between **decimal** and **float** or **double**, or from the numeric types to **char** or **bool**. Also, **char** and **bool** are not compatible with each other.

Casting Incompatible Types

Although the implicit type conversions are helpful, they will not fulfill all programming needs because they apply only to widening conversions between compatible types. For all other cases you must employ a cast. A *cast* is an instruction to the compiler to convert the outcome of an expression into a specified type. Thus, it requests an explicit type conversion. A cast has this general form:

(*target-type*) *expression*

Here, *target-type* specifies the desired type to convert the specified expression to. For example, given

```
double x, y;
```

if you want the type of the expression **x/y** to be **int**, you can write

```
(int) (x / y)
```

Here, even though **x** and **y** are of type **double**, the cast converts the outcome of the expression to **int**. The parentheses surrounding **x / y** are necessary. Otherwise, the cast to **int** would apply only to the **x** and not to the outcome of the division. The cast is necessary here because there is no implicit conversion from **double** to **int**.

When a cast involves a *narrowing conversion,* information might be lost. For example, when casting a **long** into an **int**, information will be lost if the **long**'s value is greater than the range of an **int** because its high-order bits are removed. When a floating-point value is cast to an integer type, the fractional component will also be lost due to truncation. For

example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 is lost.

The following program demonstrates some type conversions that require casts. It also shows some situations in which the casts cause data to be lost.

```
// Demonstrate casting.

using System;

class CastDemo {
  static void Main() {
    double x, y;
    byte b;
    int i;
    char ch;
    uint u;
    short s;
    long l;

    x = 10.0;
    y = 3.0;

    // Cast double to int, fractional component lost.
    i = (int) (x / y);
    Console.WriteLine("Integer outcome of x / y: " + i);
    Console.WriteLine();

    // Cast an int into a byte, no data lost.
    i = 255;
    b = (byte) i;
    Console.WriteLine("b after assigning 255: " + b +
                      " -- no data lost.");

    // Cast an int into a byte, data lost.
    i = 257;
    b = (byte) i;
    Console.WriteLine("b after assigning 257: " + b +
                      " -- data lost.");
    Console.WriteLine();

    // Cast a uint into a short, no data lost.
    u = 32000;
    s = (short) u;
    Console.WriteLine("s after assigning 32000: " + s +
                      " -- no data lost.");

    // Cast a uint into a short, data lost.
    u = 64000;
    s = (short) u;
    Console.WriteLine("s after assigning 64000: " + s +
                      " -- data lost.");
    Console.WriteLine();
```

```
    // Cast a long into a uint, no data lost.
    l = 64000;
    u = (uint) l;
    Console.WriteLine("u after assigning 64000: " + u +
                      " -- no data lost.");

    // Cast a long into a uint, data lost.
    l = -12;
    u = (uint) l;
    Console.WriteLine("u after assigning -12: " + u +
                      " -- data lost.");
    Console.WriteLine();

    // Cast an int into a char.
    b = 88; // ASCII code for X
    ch = (char) b;
    Console.WriteLine("ch after assigning 88: " + ch);
  }
}
```

The output from the program is shown here:

```
Integer outcome of x / y: 3

b after assigning 255: 255 -- no data lost.
b after assigning 257: 1 -- data lost.

s after assigning 32000: 32000 -- no data lost.
s after assigning 64000: -1536 -- data lost.

u after assigning 64000: 64000 -- no data lost.
u after assigning -12: 4294967284 -- data lost.

ch after assigning 88: X
```

Let's look at each assignment. The cast of **(x / y)** to **int** results in the truncation of the fractional component, and information is lost.

No loss of information occurs when **b** is assigned the value 255 because a **byte** can hold the value 255. However, when the attempt is made to assign **b** the value 257, information loss occurs because 257 exceeds a **byte**'s range. In both cases the casts are needed because there is no implicit conversion from **int** to **byte**.

When the **short** variable **s** is assigned the value 32,000 through the **uint** variable **u**, no data is lost because a **short** can hold the value 32,000. However, in the next assignment, **u** has the value 64,000, which is outside the range of a **short**, and data is lost. In both cases the casts are needed because there is no implicit conversion from **uint** to **short**.

Next, **u** is assigned the value 64,000 through the **long** variable **l**. In this case, no data is lost because 64,000 is within the range of a **uint**. However, when the value –12 is assigned to **u**, data is lost because a **uint** cannot hold negative numbers. In both cases the casts are needed because there is no implicit conversion from **long** to **uint**.

Finally, no information is lost, but a cast is needed when assigning a **byte** value to a **char**.

Type Conversion in Expressions

In addition to occurring within an assignment, type conversions also take place within an expression. In an expression, you can freely mix two or more different types of data as long as they are compatible with each other. For example, you can mix **short** and **long** within an expression because they are both numeric types. When different types of data are mixed within an expression, they are converted to the same type, on an operation-by-operation basis.

The conversions are accomplished through the use of C#'s *type promotion rules.* Here is the algorithm that they define for binary operations:

IF one operand is a **decimal**, THEN the other operand is promoted to **decimal** (unless it is of type **float** or **double**, in which case an error results).

ELSE IF one operand is a **double**, the second is promoted to **double**.

ELSE IF one operand is a **float**, the second is promoted to **float**.

ELSE IF one operand is a **ulong**, the second is promoted to **ulong** (unless it is of type **sbyte**, **short**, **int**, or **long**, in which case an error results).

ELSE IF one operand is a **long**, the second is promoted to **long**.

ELSE IF one operand is a **uint** and the second is of type **sbyte**, **short**, or **int**, both are promoted to **long**.

ELSE IF one operand is a **uint**, the second is promoted to **uint**.

ELSE both operands are promoted to **int**.

There are a couple of important points to be made about the type promotion rules. First, not all types can be mixed in an expression. Specifically, there is no implicit conversion from **float** or **double** to **decimal**, and it is not possible to mix **ulong** with any signed integer type. To mix these types requires the use of an explicit cast.

Second, pay special attention to the last rule. It states that if none of the preceding rules applies, then all other operands are promoted to **int**. Therefore, in an expression, all **char**, **sbyte**, **byte**, **ushort**, and **short** values are promoted to **int** for the purposes of calculation. This is called *integer promotion.* It also means that the outcome of all arithmetic operations will be no smaller than **int**.

It is important to understand that type promotions only apply to the values operated upon when an expression is evaluated. For example, if the value of a **byte** variable is promoted to **int** inside an expression, outside the expression, the variable is still a **byte**. Type promotion only affects the evaluation of an expression.

Type promotion can, however, lead to somewhat unexpected results. For example, when an arithmetic operation involves two **byte** values, the following sequence occurs. First, the **byte** operands are promoted to **int**. Then the operation takes place, yielding an **int** result. Thus, the outcome of an operation involving two **byte** values will be an **int**. This is not what you might intuitively expect. Consider the following program.

```
// A promotion surprise!

using System;

class PromDemo {
  static void Main() {
    byte b;

    b = 10;
    b = (byte) (b * b); // cast needed!!

    Console.WriteLine("b: "+ b);
  }
}
```

Somewhat counterintuitively, a cast to **byte** is needed when assigning **b * b** back to **b**! The reason is because in **b * b**, the value of **b** is promoted to **int** when the expression is evaluated. Thus, **b * b** results in an **int** value, which cannot be assigned to a **byte** variable without a cast. Keep this in mind if you get unexpected type-incompatibility error messages on expressions that would otherwise seem perfectly correct.

This same sort of situation also occurs when performing operations on **char**s. For example, in the following fragment, the cast back to **char** is needed because of the promotion of **ch1** and **ch2** to **int** within the expression

```
char ch1 = 'a', ch2 = 'b';

ch1 = (char) (ch1 + ch2);
```

Without the cast, the result of adding **ch1** to **ch2** would be **int**, which can't be assigned to a **char**.

Type promotions also occur when a unary operation, such as the unary **–**, takes place. For the unary operations, operands smaller than **int** (**byte**, **sbyte**, **short**, and **ushort**) are promoted to **int**. Also, a **char** operand is converted to **int**. Furthermore, if a **uint** value is negated, it is promoted to **long**.

Using Casts in Expressions

A cast can be applied to a specific portion of a larger expression. This gives you fine-grained control over the way type conversions occur when an expression is evaluated. For example, consider the following program. It displays the square roots of the numbers from 1 to 10. It also displays the whole number portion and the fractional part of each result, separately. To do so, it uses a cast to convert the result of **Math.Sqrt( )** to **int**.

```
// Using casts in an expression.

using System;

class CastExpr {
  static void Main() {
    double n;

     for(n = 1.0; n <= 10; n++) {
```

```
        Console.WriteLine("The square root of {0} is {1}",
                          n, Math.Sqrt(n));

        Console.WriteLine("Whole number part: {0}" ,
                          (int) Math.Sqrt(n));

        Console.WriteLine("Fractional part: {0}",
                          Math.Sqrt(n) - (int) Math.Sqrt(n) );
        Console.WriteLine();
    }
  }
}
```

Here is the output from the program:

```
The square root of 1 is 1
Whole number part: 1
Fractional part: 0

The square root of 2 is 1.4142135623731
Whole number part: 1
Fractional part: 0.414213562373095

The square root of 3 is 1.73205080756888
Whole number part: 1
Fractional part: 0.732050807568877

The square root of 4 is 2
Whole number part: 2
Fractional part: 0

The square root of 5 is 2.23606797749979
Whole number part: 2
Fractional part: 0.23606797749979

The square root of 6 is 2.44948974278318
Whole number part: 2
Fractional part: 0.449489742783178

The square root of 7 is 2.64575131106459
Whole number part: 2
Fractional part: 0.645751311064591

The square root of 8 is 2.82842712474619
Whole number part: 2
Fractional part: 0.82842712474619

The square root of 9 is 3
Whole number part: 3
Fractional part: 0

The square root of 10 is 3.16227766016838
Whole number part: 3
Fractional part: 0.16227766016838
```

As the output shows, the cast of **Math.Sqrt( )** to **int** results in the whole number component of the value. In this expression

```
Math.Sqrt(n) - (int) Math.Sqrt(n)
```

the cast to **int** obtains the whole number component, which is then subtracted from the complete value, yielding the fractional component. Thus, the outcome of the expression is **double**. Only the value of the second call to **Math.Sqrt( )** is cast to **int**.

CHAPTER 4

Operators

C# provides an extensive set of operators that give the programmer detailed control over the construction and evaluation of expressions. Most of C#'s operators fall into the following categories: *arithmetic, bitwise, relational,* and *logical.* These operators are examined in this chapter. Also discussed are the assignment operator and the **?** operator. C# also defines several other operators that handle specialized situations, such as array indexing, member access, and the lambda operator. These special operators are examined later in this book, when the features to which they apply are described.

Arithmetic Operators

C# defines the following arithmetic operators:

Operator	Meaning
+	Addition
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
– –	Decrement

The operators **+**, **–**, *****, and **/** all work in the expected way. These can be applied to any built-in numeric data type.

Although the actions of arithmetic operators are well known to all readers, a few special situations warrant some explanation. First, remember that when **/** is applied to an integer, any remainder will be truncated; for example, 10/3 will equal 3 in integer division. You can obtain the remainder of this division by using the modulus operator, **%**. The **%** is also referred to as the *remainder operator.* It yields the remainder of an integer division. For example, 10 % 3 is 1. In C#, the **%** can be applied to both integer and floating-point types.

Thus, 10.0 % 3.0 is also 1. (This differs from C/C++, which allow modulus operations only on integer types.) The following program demonstrates the modulus operator:

```
// Demonstrate the % operator.

using System;

class ModDemo {
  static void Main() {
    int iresult, irem;
    double dresult, drem;

    iresult = 10 / 3;
    irem = 10 % 3;

    dresult = 10.0 / 3.0;
    drem = 10.0 % 3.0;

    Console.WriteLine("Result and remainder of 10 / 3: " +
                      iresult + " " + irem);
    Console.WriteLine("Result and remainder of 10.0 / 3.0: " +
                      dresult + " " + drem);
  }
}
```

The output from the program is shown here:

```
Result and remainder of 10 / 3: 3 1
Result and remainder of 10.0 / 3.0: 3.33333333333333 1
```

As you can see, the % yields a remainder of 1 for both integer and floating-point operations.

Increment and Decrement

Introduced in Chapter 2, the **++** and the **– –** are the increment and decrement operators. As you will see, they have some special properties that make them quite interesting. Let's begin by reviewing precisely what the increment and decrement operators do.

The increment operator adds 1 to its operand, and the decrement operator subtracts 1. Therefore,

```
x = x + 1;
```

is the same as

```
x++;
```

and

```
x = x - 1;
```

is the same as

```
x--;
```

Understand, however, that in the increment or decrement forms, **x** is evaluated only once, not twice. This can improve efficiency in some cases.

Both the increment and decrement operators can either precede (prefix) or follow (postfix) the operand. For example

```
x = x + 1;
```

can be written as

```
++x; // prefix form
```

or as

```
x++; // postfix form
```

In the foregoing example, there is no difference whether the increment is applied as a prefix or a postfix. However, when an increment or decrement is used as part of a larger expression, there is an important difference. When an increment or decrement operator *precedes* its operand, the result of the operation is the value of the operand *after* the increment. If the operator *follows* its operand, the result of the operation is the value of the operand *before* the increment. Consider the following:

```
x = 10;
y = ++x;
```

In this case, **y** will be set to 11. This is because **x** is first incremented and then its value is returned. However, if the code is written as

```
x = 10;
y = x++;
```

then **y** will be set to 10. In this case, the value of **x** is first obtained, **x** is incremented, and then the original value of **x** is returned. In both cases, **x** is still set to 11. The difference is what is returned by the operation.

There are significant advantages in being able to control when the increment or decrement operation takes place. Consider the following program, which generates a series of numbers:

```
// Demonstrate the difference between prefix and
// postfix forms of ++.

using System;

class PrePostDemo {
  static void Main() {
    int x, y;
    int i;

    x = 1;
    Console.WriteLine("Series generated using y = x + x++;");
    for(i = 0; i < 10; i++) {

      y = x + x++; // postfix ++
```

```
      Console.WriteLine(y + " ");
    }
    Console.WriteLine();

    x = 1;
    Console.WriteLine("Series generated using y = x + ++x;");
    for(i = 0; i < 10; i++) {

      y = x + ++x; // prefix ++

      Console.WriteLine(y + " ");
    }
    Console.WriteLine();

  }
}
```

The output is shown here:

```
Series generated using y = x + x++;
2
4
6
8
10
12
14
16
18
20

Series generated using y = x + ++x;
3
5
7
9
11
13
15
17
19
21
```

As the output confirms, the statement

```
y = x + x++;
```

adds the original value of **x** to **x** and assigns this result to **y**. The value of **x** is incremented after its value has been obtained. However, the statement

```
y = x + ++x;
```

obtains the value of **x**, increments **x**, and then adds that value to the original value of **x**. The result is assigned to **y**. As the output shows, simply changing **++x** to **x++** changes the number series from even to odd.

One other point about the preceding example: Don't let expressions like

```
x + ++x
```

intimidate you. Although having two operators back-to-back is a bit unsettling at first glance, the compiler keeps it all straight. Just remember, this expression simply adds the value of **x** to the value of **x** incremented.

Relational and Logical Operators

In the terms *relational operator* and *logical operator, relational* refers to the relationships that values can have with one another, and *logical* refers to the ways in which true and false values can be connected together. Since the relational operators produce true or false results, they often work with the logical operators. For this reason they will be discussed together here.

The relational operators are as follows:

Operator	Meaning
= =	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The logical operators are shown next:

Operator	Meaning
&	AND
|	OR
^	XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	NOT

The outcome of the relational and logical operators is a **bool** value.

In general, objects can be compared for equality or inequality using == and **!=**. However, the comparison operators, **<**, **>**, **<=**, or **>=**, can be applied only to those types that support an ordering relationship. Therefore, all of the relational operators can be applied to all numeric types. However, values of type **bool** can only be compared for equality or inequality since the **true** and **false** values are not ordered. For example, **true > false** has no meaning in C#.

For the logical operators, the operands must be of type **bool**, and the result of a logical operation is of type **bool**. The logical operators, **&**, **|**, **^**, and **!**, support the basic logical operations AND, OR, XOR, and NOT, according to the following truth table:

p	q	p & q	p | q	p ^ q	!p
False	False	False	False	False	True
True	False	False	True	True	False
False	True	False	True	True	True
True	True	True	True	False	False

As the table shows, the outcome of an exclusive OR operation is true when one and only one operand is true.

Here is a program that demonstrates several of the relational and logical operators:

```
// Demonstrate the relational and logical operators.

using System;

class RelLogOps {
  static void Main() {
    int i, j;
    bool b1, b2;

    i = 10;
    j = 11;
    if(i < j) Console.WriteLine("i < j");
    if(i <= j) Console.WriteLine("i <= j");
    if(i != j) Console.WriteLine("i != j");
    if(i == j) Console.WriteLine("this won't execute");
    if(i >= j) Console.WriteLine("this won't execute");
    if(i > j) Console.WriteLine("this won't execute");

    b1 = true;
    b2 = false;
    if(b1 & b2) Console.WriteLine("this won't execute");
    if(!(b1 & b2)) Console.WriteLine("!(b1 & b2) is true");
    if(b1 | b2) Console.WriteLine("b1 | b2 is true");
    if(b1 ^ b2) Console.WriteLine("b1 ^ b2 is true");
  }
}
```

The output from the program is shown here:

```
i < j
i <= j
i != j
!(b1 & b2) is true
b1 | b2 is true
b1 ^ b2 is true
```

The logical operators provided by C# perform the most commonly used logical operations. However, several other operations are defined by the rules for formal logic. These other logical operations can be constructed using the logical operators supported by C#. Thus, C# supplies a set of logical operators sufficient to construct any other logical operation. For example, another logical operation is *implication.* Implication is a binary operation in which the outcome is false only when the left operand is true and the right operand is false. (The implication operation reflects the idea that true cannot imply false.) Thus, the truth table for the implication operator is shown here:

p	q	p implies q
True	True	True
True	False	False
False	False	True
False	True	True

The implication operation can be constructed using a combination of the **!** and the **|** operator, as shown here:

```
!p | q
```

The following program demonstrates this implementation:

```
// Create an implication operator in C#.

using System;

class Implication {
  static void Main() {
    bool p=false, q=false;
    int i, j;

    for(i = 0; i < 2; i++) {
      for(j = 0; j < 2; j++) {
        if(i==0) p = true;
        if(i==1) p = false;
        if(j==0) q = true;
        if(j==1) q = false;

        Console.WriteLine("p is " + p + ", q is " + q);
        if(!p | q) Console.WriteLine(p + " implies " + q +
                    " is " + true);
        Console.WriteLine();
      }
    }
  }
}
```

The output is shown here:

```
p is True, q is True
True implies True is True

p is True, q is False
```

```
p is False, q is True
False implies True is True

p is False, q is False
False implies False is True
```

Short-Circuit Logical Operators

C# supplies special *short-circuit* versions of its AND and OR logical operators that can be used to produce more efficient code. To understand why, consider the following. In an AND operation, if the first operand is false, then the outcome is false no matter what value the second operand has. In an OR operation, if the first operand is true, then the outcome of the operation is true no matter what the value of the second operand. Thus, in these two cases there is no need to evaluate the second operand. By not evaluating the second operand, time is saved and more efficient code is produced.

The short-circuit AND operator is **&&** and the short-circuit OR operator is **||**. As described earlier, their normal counterparts are **&** and **|**. The only difference between the normal and short-circuit versions is that the normal operands will always evaluate each operand, but short-circuit versions will evaluate the second operand only when necessary.

Here is a program that demonstrates the short-circuit AND operator. The program determines if the value in **d** is a factor of **n**. It does this by performing a modulus operation. If the remainder of **n / d** is zero, then **d** is a factor. However, since the modulus operation involves a division, the short-circuit form of the AND is used to prevent a divide-by-zero error.

```
// Demonstrate the short-circuit operators.

using System;

class SCops {
  static void Main() {
    int n, d;

    n = 10;
    d = 2;
    if(d != 0 && (n % d) == 0)
      Console.WriteLine(d + " is a factor of " + n);

    d = 0; // now, set d to zero

    // Since d is zero, the second operand is not evaluated.
    if(d != 0 && (n % d) == 0)
      Console.WriteLine(d + " is a factor of " + n);

    // Now, try the same thing without short-circuit operator.
    // This will cause a divide-by-zero error.
    if(d != 0 & (n % d) == 0)
      Console.WriteLine(d + " is a factor of " + n);
  }
}
```

To prevent a divide-by-zero error, the **if** statement first checks to see if **d** is equal to zero. If it is, then the short-circuit AND stops at that point and does not perform the modulus

division. Thus, in the first test, **d** is 2 and the modulus operation is performed. The second test fails because **d** is set to zero, and the modulus operation is skipped, avoiding a divide-by-zero error. Finally, the normal AND operator is tried. This causes both operands to be evaluated, which leads to a runtime error when the division-by-zero occurs.

Since the short-circuit operators are, in some cases, more efficient than their normal counterparts, you might be wondering why C# still offers the normal AND and OR operators. The answer is that in some cases you will want both operands of an AND or OR operation to be evaluated because of the side effects produced. Consider the following:

```
// Side effects can be important.

using System;

class SideEffects {
  static void Main() {
    int i;
    bool someCondition = false;

    i = 0;

    // Here, i is still incremented even though the if statement fails.
    if(someCondition & (++i < 100))
       Console.WriteLine("this won't be displayed");
    Console.WriteLine("if statement executed: " + i); // displays 1

    // In this case, i is not incremented because the short-circuit
    // operator skips the increment.
    if(someCondition && (++i < 100))
      Console.WriteLine("this won't be displayed");
    Console.WriteLine("if statement executed: " + i); // still 1 !!
  }
}
```

First, notice that the **bool** variable **someCondition** is initialized to **false**. Next, examine each **if** statement. As the comments indicate, in the first **if** statement, **i** is incremented despite the fact that **someCondition** is false. When the **&** is used, as it is in the first **if** statement, the expression on the right side of the **&** is evaluated no matter what value the expression on the left has. However, in the second **if** statement, the short-circuit operator is used. In this case, the variable **i** is not incremented because the left operand, **someCondition**, is false, which causes the expression on the right to be skipped. The lesson here is that if your code expects the right-hand operand of an AND or OR operation to be evaluated, then you must use C#'s non-short-circuit forms for these operations.

One other point: The short-circuit AND is also known as the *conditional AND,* and the short-circuit OR is also called the *conditional OR.*

The Assignment Operator

The *assignment operator* is the single equal sign, **=**. The assignment operator works in C# much as it does in other computer languages. It has this general form:

var-name = *expression*;

Here, the type of *var-name* must be compatible with the type of *expression.*

The assignment operator does have one interesting attribute that you may not be familiar with: It allows you to create a chain of assignments. For example, consider this fragment:

```
int x, y, z;

x = y = z = 100; // set x, y, and z to 100
```

This fragment sets the variables **x**, **y**, and **z** to 100 using a single statement. This works because the **=** is an operator that yields the assigned value. Thus, the value of **z = 100** is 100, which is then assigned to **y**, which in turn is assigned to **x**. Using a "chain of assignment" is an easy way to set a group of variables to a common value.

Compound Assignments

C# provides special compound assignment operators that simplify the coding of certain assignment statements. Let's begin with an example. The assignment statement shown here:

```
x = x + 10;
```

can be written using a compound assignment as

```
x += 10;
```

The operator pair **+=** tells the compiler to assign to **x** the value of **x** plus 10.

Here is another example. The statement

```
x = x - 100;
```

is the same as

```
x -= 100;
```

Both statements assign to **x** the value of **x** minus 100.

There are compound assignment operators for many of the binary operators (that is, those that require two operands). The general form of the shorthand is

var-name op = expression;

Thus, the arithmetic and logical assignment operators are

+=	–=	*=	/=
%=	&=	|=	^=

Because the compound assignment statements are shorter than their noncompound equivalents, the compound assignment operators are also sometimes called the *shorthand assignment* operators.

The compound assignment operators provide two benefits. First, they are more compact than their "longhand" equivalents. Second, they can result in more efficient executable code (because the left-hand operand is evaluated only once). For these reasons, you will often see the compound assignment operators used in professionally written C# programs.

The Bitwise Operators

C# provides a set of *bitwise* operators that expand the types of problems to which C# can be applied. The bitwise operators act directly upon the bits of their operands. They are defined only for integer operands. They cannot be used on **bool**, **float**, or **double**.

They are called the *bitwise* operators because they are used to test, set, or shift the bits that comprise an integer value. Among other uses, bitwise operations are important to a wide variety of systems-level programming tasks, such as analyzing status information from a device. Table 4-1 lists the bitwise operators.

The Bitwise AND, OR, XOR, and NOT Operators

The bitwise operators AND, OR, XOR, and NOT are **&**, **|**, **^**, and **~**. They perform the same operations as their Boolean logic equivalents described earlier. The difference is that the bitwise operators work on a bit-by-bit basis. The following table shows the outcome of each operation using 1s and 0s:

p	q	p & q	p | q	p ^ q	~p
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

In terms of its most common usage, you can think of the bitwise AND as a way to turn bits off. That is, any bit that is 0 in either operand will cause the corresponding bit in the outcome to be set to 0. For example

```
  1101 0011
& 1010 1010
  ----------
  1000 0010
```

Operator	Result
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR (XOR)
>>	Shift right
<<	Shift left
~	One's complement (unary NOT)

TABLE 4-1 The Bitwise Operators

The following program demonstrates the **&** by using it to convert odd numbers into even numbers. It does this by turning off bit zero. For example, the low-order byte of the number 9 in binary is 0000 1001. When bit zero is turned off, this number becomes 8, or 0000 1000 in binary.

```
// Use bitwise AND to make a number even.

using System;

class MakeEven {
  static void Main() {
    ushort num;
    ushort i;

    for(i = 1; i <= 10; i++) {
      num = i;

      Console.WriteLine("num: " + num);

      num = (ushort) (num & 0xFFFE);

      Console.WriteLine("num after turning off bit zero: "
                        +  num + "\n");
    }
  }
}
```

The output from this program is shown here:

```
num: 1
num after turning off bit zero: 0

num: 2
num after turning off bit zero: 2

num: 3
num after turning off bit zero: 2

num: 4
num after turning off bit zero: 4

num: 5
num after turning off bit zero: 4

num: 6
num after turning off bit zero: 6

num: 7
num after turning off bit zero: 6

num: 8
num after turning off bit zero: 8

num: 9
num after turning off bit zero: 8
```

```
num: 10
num after turning off bit zero: 10
```

The value **0xFFFE** used in the AND statement is the hexadecimal representation of 1111 1111 1111 1110. Therefore, the AND operation leaves all bits in **num** unchanged except for bit zero, which is set to zero. Thus, even numbers are unchanged, but odd numbers are made even by reducing their value by 1.

The AND operator is also useful when you want to determine whether a bit is on or off. For example, this program determines if a number is odd:

```
//  Use bitwise AND to determine if a number is odd.

using System;

class IsOdd {
  static void Main() {
    ushort num;

    num = 10;

    if((num & 1) == 1)
      Console.WriteLine("This won't display.");

    num = 11;

    if((num & 1) == 1)
      Console.WriteLine(num + " is odd.");

  }
}
```

The output is shown here:

```
11 is odd.
```

In the **if** statements, the value of **num** is ANDed with 1. If bit zero in **num** is set, the result of **num & 1** is 1; otherwise, the result is zero. Therefore, the **if** statement can succeed only when the number is odd.

You can use the bit-testing capability of the bitwise **&** to create a program that uses the bitwise **&** to show the bits of a **byte** value in binary format. Here is one approach:

```
// Display the bits within a byte.

using System;

class ShowBits {
  static void Main() {
    int t;
    byte val;

    val = 123;
    for(t=128; t > 0; t = t/2) {
      if((val & t) != 0) Console.Write("1 ");
```

```
      if((val & t) == 0) Console.Write("0 ");
    }
  }
}
```

The output is shown here:

```
0 1 1 1 1 0 1 1
```

The **for** loop successively tests each bit in **val**, using the bitwise AND, to determine if it is on or off. If the bit is on, the digit **1** is displayed; otherwise, **0** is displayed.

The bitwise OR can be used to turn bits on. Any bit that is set to 1 in either operand will cause the corresponding bit in the variable to be set to 1. For example

```
  1101 0011
| 1010 1010
  -------------------
  1111 1011
```

You can make use of the OR to change the make-even program shown earlier into a make-odd program, as shown here:

```
//  Use bitwise OR to make a number odd.

using System;

class MakeOdd {
  static void Main() {
    ushort num;
    ushort i;

    for(i = 1; i <= 10; i++) {
      num = i;

      Console.WriteLine("num: " + num);

      num = (ushort) (num | 1);

      Console.WriteLine("num after turning on bit zero: "
                        +  num + "\n");
    }
  }
}
```

The output from this program is shown here:

```
num: 1
num after turning on bit zero: 1

num: 2
num after turning on bit zero: 3

num: 3
num after turning on bit zero: 3
```

```
num: 4
num after turning on bit zero: 5

num: 5
num after turning on bit zero: 5

num: 6
num after turning on bit zero: 7

num: 7
num after turning on bit zero: 7

num: 8
num after turning on bit zero: 9

num: 9
num after turning on bit zero: 9

num: 10
num after turning on bit zero: 11
```

The program works by ORing each number with the value 1, because 1 is the value that produces a value in binary in which only bit zero is set. When this value is ORed with any other value, it produces a result in which the low-order bit is set and all other bits remain unchanged. Thus, a value that is even will be increased by 1, becoming odd.

An exclusive OR, usually abbreviated XOR, will set a bit on if, and only if, the bits being compared are different, as illustrated here:

```
   0 1 1 1 1 1 1 1
^  1 0 1 1 1 0 0 1
  -----------------------
   1 1 0 0 0 1 1 0
```

The XOR operator has an interesting property that is useful in a variety of situations. When some value X is XORed with another value Y, and then that result is XORed with Y again, X is produced. That is, given the sequence

```
R1 = X ^ Y;
R2 = R1 ^ Y;
```

R2 is the same value as X. Thus, the outcome of a sequence of two XORs using the same value produces the original value. This feature of the XOR can be put into action to create a simple cipher in which some integer is the key that is used to both encode and decode a message by XORing the characters in that message. To encode, the XOR operation is applied the first time, yielding the ciphertext. To decode, the XOR is applied a second time, yielding the plaintext. Of course, such a cipher has no practical value, being trivially easy to break. It does, however, provide an interesting way to demonstrate the effects of the XOR, as the following program shows:

```
// Demonstrate the XOR.

using System;
```

```
class Encode {
  static void Main() {
    char ch1 = 'H';
    char ch2 = 'i';
    char ch3 = '!';
    int key = 88;

    Console.WriteLine("Original message: " + ch1 + ch2 + ch3);

    // Encode the message.
    ch1 = (char) (ch1 ^ key);
    ch2 = (char) (ch2 ^ key);
    ch3 = (char) (ch3 ^ key);

    Console.WriteLine("Encoded message: " + ch1 + ch2 + ch3);

    // Decode the message.
    ch1 = (char) (ch1 ^ key);
    ch2 = (char) (ch2 ^ key);
    ch3 = (char) (ch3 ^ key);

    Console.WriteLine("Encoded message: " + ch1 + ch2 + ch3);
  }
}
```

Here is the output:

```
Original message: Hi!
Encoded message: ❑1y
Encoded message: Hi!
```

As you can see, the result of two XORs using the same key produces the decoded message. (Remember, this simple XOR cipher is not suitable for any real-world, practical use because it is inherently insecure.)

The unary one's complement (NOT) operator reverses the state of all the bits of the operand. For example, if some integer called **A** has the bit pattern 1001 0110, then **~A** produces a result with the bit pattern 0110 1001.

The following program demonstrates the NOT operator by displaying a number and its complement in binary:

```
// Demonstrate the bitwise NOT.

using System;

class NotDemo {
  static void Main() {
    sbyte b = -34;

    for(int t=128; t > 0; t = t/2) {
      if((b & t) != 0) Console.Write("1 ");
      if((b & t) == 0) Console.Write("0 ");
    }
    Console.WriteLine();
```

```
    // reverse all bits
    b = (sbyte) ~b;

    for(int t=128; t > 0; t = t/2) {
      if((b & t) != 0) Console.Write("1 ");
      if((b & t) == 0) Console.Write("0 ");
    }
  }
}
```

Here is the output:

```
1 1 0 1 1 1 1 0
0 0 1 0 0 0 0 1
```

The Shift Operators

In C# it is possible to shift the bits that comprise an integer value to the left or to the right by a specified amount. C# defines the two bit-shift operators shown here:

<<	Left shift
>>	Right shift

The general forms for these operators are shown here:

value << *num-bits*
value >> *num-bits*

Here, *value* is the value being shifted by the number of bit positions specified by *num-bits.*

A left shift causes all bits within the specified value to be shifted left one position and a zero bit to be brought in on the right. A right shift causes all bits to be shifted right one position. In the case of a right shift on an unsigned value, a zero is brought in on the left. In the case of a right shift on a signed value, the sign bit is preserved. Recall that negative numbers are represented by setting the high-order bit of an integer value to 1. Thus, if the value being shifted is negative, each right shift brings in a 1 on the left. If the value is positive, each right shift brings in a 0 on the left.

For both left and right shifts, the bits shifted out are lost. Thus, a shift is not a rotate and there is no way to retrieve a bit that has been shifted out.

Here is a program that graphically illustrates the effect of a left and right shift. Here, an integer is given an initial value of 1, which means that its low-order bit is set. Then, eight shifts are performed on the integer. After each shift, the lower eight bits of the value are shown. The process is then repeated, except that a 1 is put in the eighth bit position, and right shifts are performed.

```
// Demonstrate the shift << and >> operators.

using System;

class ShiftDemo {
  static void Main() {
    int val = 1;
```

```
    for(int i = 0; i < 8; i++) {
      for(int t=128; t > 0; t = t/2) {
        if((val & t) != 0) Console.Write("1 ");
        if((val & t) == 0) Console.Write("0 ");
      }
      Console.WriteLine();
      val = val << 1; // left shift
    }
    Console.WriteLine();

    val = 128;
    for(int i = 0; i < 8; i++) {
      for(int t=128; t > 0; t = t/2) {
        if((val & t) != 0) Console.Write("1 ");
        if((val & t) == 0) Console.Write("0 ");
      }
      Console.WriteLine();
      val = val >> 1; // right shift
    }
  }
}
```

The output from the program is shown here:

```
0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0
0 0 0 0 1 0 0 0
0 0 0 1 0 0 0 0
0 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0

1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1
```

Since binary is based on powers of 2, the shift operators can be used as a way to multiply or divide an integer by 2. A shift left doubles a value. A shift right halves it. Of course, this works only as long as you are not shifting bits off one end or the other. Here is an example:

```
// Use the shift operators to multiply and divide by 2.

using System;

class MultDiv {
  static void Main() {
    int n;

    n = 10;
```

```
    Console.WriteLine("Value of n: " + n);

    // Multiply by 2.
    n = n << 1;
    Console.WriteLine("Value of n after n = n * 2: " + n);

    // Multiply by 4.
    n = n << 2;
    Console.WriteLine("Value of n after n = n * 4: " + n);

    // Divide by 2.
    n = n >> 1;
    Console.WriteLine("Value of n after n = n / 2: " + n);

    // Divide by 4.
    n = n >> 2;
    Console.WriteLine("Value of n after n = n / 4: " + n);
    Console.WriteLine();

    // Reset n.
    n = 10;
    Console.WriteLine("Value of n: " + n);

    // Multiply by 2, 30 times.
    n = n << 30; // data is lost
    Console.WriteLine("Value of n after left-shifting 30 places: " + n);
  }
}
```

The output is shown here:

```
Value of n: 10
Value of n after n = n * 2: 20
Value of n after n = n * 4: 80
Value of n after n = n / 2: 40
Value of n after n = n / 4: 10

Value of n: 10
Value of n after left-shifting 30 places: -2147483648
```

Notice the last line in the output. When the value 10 is left-shifted 30 times, information is lost because bits are shifted out of the range of an **int**. In this case, the garbage value produced is negative because a 1 bit is shifted into the high-order bit, which is used as a sign bit, causing the number to be interpreted as negative. This illustrates why you must be careful when using the shift operators to multiply or divide a value by 2. (See Chapter 3 for an explanation of signed vs. unsigned data types.)

Bitwise Compound Assignments

All of the binary bitwise operators can be used in compound assignments. For example, the following two statements both assign to **x** the outcome of an XOR of **x** with the value 127:

```
x = x ^ 127;
x ^= 127;
```

The ? Operator

One of C#'s most fascinating operators is the **?**, which is C#'s conditional operator. The **?** operator is often used to replace certain types of if-then-else constructions. The **?** is called a *ternary operator* because it requires three operands. It takes the general form

Exp1 ? *Exp2* : *Exp3*;

where *Exp1* is a **bool** expression, and *Exp2* and *Exp3* are expressions. The type of *Exp2* and *Exp3* must be the same. Notice the use and placement of the colon.

The value of a **?** expression is determined like this: *Exp1* is evaluated. If it is true, then *Exp2* is evaluated and becomes the value of the entire **?** expression. If *Exp1* is false, then *Exp3* is evaluated, and its value becomes the value of the expression. Consider this example, which assigns **absval** the absolute value of **val**:

```
absval = val < 0 ? -val : val; // get absolute value of val
```

Here, **absval** will be assigned the value of **val** if **val** is zero or greater. If **val** is negative, then **absval** will be assigned the negative of that value (which yields a positive value).

Here is another example of the **?** operator. This program divides two numbers, but will not allow a division by zero.

```
// Prevent a division by zero using the ?.

using System;

class NoZeroDiv {
  static void Main() {
    int result;

    for(int i = -5; i < 6; i++) {
      result = i != 0 ? 100 / i : 0;
      if(i != 0)
        Console.WriteLine("100 / " + i + " is " + result);
    }
  }
}
```

The output from the program is shown here:

```
100 / -5 is -20
100 / -4 is -25
100 / -3 is -33
100 / -2 is -50
100 / -1 is -100
100 / 1 is 100
100 / 2 is 50
100 / 3 is 33
100 / 4 is 25
100 / 5 is 20
```

Pay special attention to this line from the program:

```
result = i != 0 ? 100 / i : 0;
```

Here, **result** is assigned the outcome of the division of 100 by **i**. However, this division takes place only if **i** is not 0. When **i** is 0, a placeholder value of 0 is assigned to **result**.

You don't actually have to assign the value produced by the **?** to some variable. For example, you could use the value as an argument in a call to a method. Or, if the expressions are all of type **bool**, the **?** can be used as the conditional expression in a loop or **if** statement. For example, the following program displays the results of dividing 100 by only even, non-zero values:

```
// Divide by only even, non-zero values.

using System;

class NoZeroDiv2 {
  static void Main() {

    for(int i = -5; i < 6; i++)
      if(i != 0 ? (i%2 == 0) : false)
        Console.WriteLine("100 / " + i + " is " + 100 / i);
  }
}
```

Notice the **if** statement. If **i** is zero, then the outcome of the **if** is false. Otherwise, if **i** is non-zero, then the outcome of the **if** is true if **i** is even and false if **i** is odd. Thus, only even, non-zero divisors are allowed. Although this example is somewhat contrived for the sake of illustration, such constructs are occasionally very useful.

Spacing and Parentheses

An expression in C# can have tabs and spaces in it to make it more readable. For example, the following two expressions are the same, but the second is easier to read:

```
x=10/y*(127+x);

x = 10 / y * (127 + x);
```

Parentheses can be used to group subexpressions, thereby effectively increasing the precedence of the operations contained within them, just like in algebra. Use of redundant or additional parentheses will not cause errors or slow down execution of the expression. You are encouraged to use parentheses to make clear the exact order of evaluation, both for yourself and for others who may have to figure out your program later. For example, which of the following two expressions is easier to read?

```
x = y/3-34*temp+127;

x = (y/3) - (34*temp) + 127;
```

Operator Precedence

Table 4-2 shows the order of precedence for all C# operators, from highest to lowest. This table includes several operators that will be discussed later in this book.

Highest									
()	[]	.	++ (postfix)	– – (postfix)	checked	new	sizeof	typeof	unchecked
!	~	(cast)	+ (unary)	– (unary)	++ (prefix)	– – (prefix)			
*	/	%							
+	–								
<<	>>								
<	>	<=	>=	is					
==	!=								
&									
^									
|									
&&									
||									
??									
?:									
=	op=	=>							
Lowest									

TABLE 4-2 The Precedence of the C# Operators

CHAPTER 5

Program Control Statements

This chapter discusses C#'s program control statements. There are three categories of program control statements: *selection* statements, which are the **if** and the **switch**; *iteration* statements, which consist of the **for**, **while**, **do-while**, and **foreach** loops; and *jump* statements, which include **break**, **continue**, **goto**, **return**, and **throw**. Except for **throw**, which is part of C#'s exception-handling mechanism and is discussed in Chapter 13, the others are examined here.

The if Statement

Chapter 2 introduced the **if** statement. It is examined in detail here. The complete form of the **if** statement is

```
if(condition) statement;
else statement;
```

where the targets of the **if** and **else** are single statements. The **else** clause is optional. The targets of both the **if** and **else** can be blocks of statements. The general form of the **if** using blocks of statements is

```
if(condition)
{
  statement sequence
}
else
{
  statement sequence
}
```

If the conditional expression is true, the target of the **if** will be executed; otherwise, if it exists, the target of the **else** will be executed. At no time will both of them be executed. The conditional expression controlling the **if** must produce a **bool** result.

Here is a simple example that uses an **if** and **else** to report if a number is positive or negative:

```
// Determine if a value is positive or negative.

using System;
```

```
class PosNeg {
  static void Main() {
    int i;

    for(i=-5; i <= 5; i++) {
      Console.Write("Testing " + i + ": ");

      if(i < 0) Console.WriteLine("negative");
      else Console.WriteLine("positive");
    }
  }
}
```

The output is shown here:

```
Testing -5: negative
Testing -4: negative
Testing -3: negative
Testing -2: negative
Testing -1: negative
Testing 0: positive
Testing 1: positive
Testing 2: positive
Testing 3: positive
Testing 4: positive
Testing 5: positive
```

In this example, if **i** is less than zero, then the target of the **if** is executed. Otherwise, the target of the **else** is executed. In no case are both executed.

Nested ifs

A *nested if* is an **if** statement that is the target of another **if** or **else**. Nested **if**s are very common in programming. The main thing to remember about nested **if**s in C# is that an **else** clause always refers to the nearest **if** statement that is within the same block as the **else** and not already associated with an **else**. Here is an example:

```
if(i == 10) {
  if(j < 20) a = b;
  if(k > 100) c = d;
  else a = c; // this else refers to if(k > 100)
}
else a = d; // this else refers to if(i == 10)
```

As the comments indicate, the final **else** is not associated with **if(j < 20)** because it is not in the same block (even though it is the nearest **if** without an **else**). Rather, the final **else** is associated with **if(i == 10)**. The inner **else** refers to **if(k > 100)** because it is the closest **if** within the same block.

The following program demonstrates a nested **if**. In the positive/negative program shown earlier, zero is reported as positive. However, as a general rule, zero is considered signless. The following version of the program reports zero as being neither positive nor negative.

```
// Determine if a value is positive, negative, or zero.

using System;

class PosNegZero {
  static void Main() {
    int i;

    for(i=-5; i <= 5; i++) {

      Console.Write("Testing " + i + ": ");

      if(i < 0) Console.WriteLine("negative");
      else if(i == 0) Console.WriteLine("no sign");
        else Console.WriteLine("positive");
    }
  }
}
```

Here is the output:

```
Testing -5: negative
Testing -4: negative
Testing -3: negative
Testing -2: negative
Testing -1: negative
Testing 0: no sign
Testing 1: positive
Testing 2: positive
Testing 3: positive
Testing 4: positive
Testing 5: positive
```

The if-else-if Ladder

A common programming construct that is based upon the nested **if** is the ***if-else-if*** *ladder*. It looks like this:

```
if(condition)
  statement;
else if(condition)
  statement;
else if(condition)
  statement;
.
.
.
else
  statement;
```

The conditional expressions are evaluated from the top downward. As soon as a true condition is found, the statement associated with it is executed, and the rest of the ladder is bypassed.

If none of the conditions is true, then the final **else** clause will be executed. The final **else** often acts as a default condition. That is, if all other conditional tests fail, then the last **else** clause is executed. If there is no final **else** and all other conditions are false, then no action will take place.

The following program demonstrates the **if-else-if** ladder. It finds the smallest single-digit factor (other than 1) for a given value.

```
// Determine smallest single-digit factor.

using System;

class Ladder {
  static void Main() {
    int num;

    for(num = 2; num < 12; num++) {
      if((num % 2) == 0)
        Console.WriteLine("Smallest factor of " + num + " is 2.");
      else if((num % 3) == 0)
        Console.WriteLine("Smallest factor of " + num + " is 3.");
      else if((num % 5) == 0)
        Console.WriteLine("Smallest factor of " + num + " is 5.");
      else if((num % 7) == 0)
        Console.WriteLine("Smallest factor of " + num + " is 7.");
      else
        Console.WriteLine(num + " is not divisible by 2, 3, 5, or 7.");
    }
  }
}
```

The program produces the following output:

```
Smallest factor of 2 is 2.
Smallest factor of 3 is 3.
Smallest factor of 4 is 2.
Smallest factor of 5 is 5.
Smallest factor of 6 is 2.
Smallest factor of 7 is 7.
Smallest factor of 8 is 2.
Smallest factor of 9 is 3.
Smallest factor of 10 is 2.
11 is not divisible by 2, 3, 5, or 7.
```

As you can see, the **else** is executed only if none of the preceding **if** statements succeeds.

The switch Statement

The second of C#'s selection statements is **switch**. The **switch** provides for a multiway branch. Thus, it enables a program to select among several alternatives. Although a series of nested **if** statements can perform multiway tests, for many situations the **switch** is a more efficient approach. It works like this: The value of an expression is successively tested against a list of constants. When a match is found, the statement sequence associated with that match is executed. The general form of the **switch** statement is

```
switch(expression) {
   case constant1:
      statement sequence
      break;
   case constant2:
      statement sequence
      break;
   case constant3:
      statement sequence
      break;
  .
  .
  .
   default:
     statement sequence
     break;
}
```

The **switch** expression must be of an integer type, such as **char**, **byte**, **short**, or **int**, of an enumeration type, or of type **string**. (Enumerations and the **string** type are described later in this book.) Thus, floating-point expressions, for example, are not allowed. Frequently, the expression controlling the **switch** is simply a variable. The **case** constants must be of a type compatible with the expression. No two **case** constants in the same **switch** can have identical values.

The **default** sequence is executed if no **case** constant matches the expression. The **default** is optional; if it is not present, no action takes place if all matches fail. When a match is found, the statements associated with that **case** are executed until the **break** is encountered.

The following program demonstrates the **switch**:

```
// Demonstrate the switch.

using System;

class SwitchDemo {
  static void Main() {
    int i;

    for(i=0; i<10; i++)
      switch(i) {
        case 0:
          Console.WriteLine("i is zero");
          break;
        case 1:
          Console.WriteLine("i is one");
          break;
        case 2:
          Console.WriteLine("i is two");
          break;
        case 3:
          Console.WriteLine("i is three");
          break;
```

```
        case 4:
          Console.WriteLine("i is four");
          break;
        default:
          Console.WriteLine("i is five or more");
          break;
      }
  }
}
```

The output produced by this program is shown here:

```
i is zero
i is one
i is two
i is three
i is four
i is five or more
i is five or more
i is five or more
i is five or more
i is five or more
```

As you can see, each time through the loop, the statements associated with the **case** constant that matches **i** are executed. All others are bypassed. When **i** is five or greater, no **case** constants match, so the **default** is executed.

In the preceding example, the **switch** was controlled by an **int** variable. As explained, you can control a **switch** with any integer type, including **char**. Here is an example that uses a **char** expression and **char case** constants:

```
// Use a char to control the switch.

using System;

class SwitchDemo2 {
  static void Main() {
    char ch;

    for(ch='A'; ch<= 'E'; ch++)
      switch(ch) {
        case 'A':
          Console.WriteLine("ch is A");
          break;
        case 'B':
          Console.WriteLine("ch is B");
          break;
        case 'C':
          Console.WriteLine("ch is C");
          break;
        case 'D':
          Console.WriteLine("ch is D");
          break;
        case 'E':
          Console.WriteLine("ch is E");
```

```
          break;
      }
  }
}
```

The output from this program is shown here:

```
ch is A
ch is B
ch is C
ch is D
ch is E
```

Notice that this example does not include a **default** case. Remember, the **default** is optional. When not needed, it can be left out.

In C#, it is an error for the statement sequence associated with one **case** to continue on into the next **case**. This is called the "no fall-through" rule. This is why **case** sequences end with a **break** statement. (You can avoid fall-through in other ways, such as by using the **goto** discussed later in this chapter, but **break** is by far the most commonly used approach.) When encountered within the statement sequence of a **case**, the **break** statement causes program flow to exit from the entire **switch** statement and resume at the next statement outside the **switch**. The **default** sequence also must not "fall through," and it too usually ends with **break**.

The no fall-through rule is one point on which C# differs from C, C++, and Java. In those languages, one **case** may continue on (that is, fall through) into the next **case**. There are two reasons that C# instituted the no fall-through rule for **case**s: First, it allows the compiler to freely rearrange the order of the **case** sequences, perhaps for purposes of optimization. Such a rearrangement would not be possible if one **case** could flow into the next. Second, requiring each **case** to explicitly end prevents a programmer from accidentally allowing one **case** to flow into the next.

Although you cannot allow one **case** sequence to fall through into another, you can have two or more **case** labels refer to the same code sequence, as shown in this example:

```
// Empty cases can fall through.

using System;

class EmptyCasesCanFall {
  static void Main() {
    int i;

    for(i=1; i < 5; i++)
      switch(i) {
        case 1:
        case 2:
        case 3: Console.WriteLine("i is 1, 2 or 3");
          break;
        case 4: Console.WriteLine("i is 4");
          break;
      }

  }
}
```

The output is shown here:

```
i is 1, 2 or 3
i is 1, 2 or 3
i is 1, 2 or 3
i is 4
```

In this example, if **i** has the value 1, 2, or 3, then the first **WriteLine( )** statement executes. If **i** is 4, then the second **WriteLine( )** statement executes. The stacking of **case**s does not violate the no fall-through rule, because the **case** statements all use the same statement sequence.

Stacking **case** labels is a commonly employed technique when several **case**s share common code. This technique prevents the unnecessary duplication of code sequences.

Nested switch Statements

It is possible to have a **switch** as part of the statement sequence of an outer **switch**. This is called a *nested* ***switch.*** The **case** constants of the inner and outer **switch** can contain common values and no conflicts will arise. For example, the following code fragment is perfectly acceptable:

```
switch(ch1) {
  case 'A': Console.WriteLine("This A is part of outer switch.");
    switch(ch2) {
      case 'A':
        Console.WriteLine("This A is part of inner switch");
        break;
      case 'B': // ...
    } // end of inner switch
    break;
  case 'B': // ...
```

The for Loop

The **for** loop was introduced in Chapter 2. Here, it is examined in detail. You might be surprised at just how powerful and flexible the **for** loop is. Let's begin by reviewing the basics, starting with the most traditional forms of the **for**.

The general form of the **for** loop for repeating a single statement is

for(*initialization*; *condition*; *iteration*) *statement*;

For repeating a block, the general form is

```
for(initialization; condition; iteration)
{
  statement sequence
}
```

The *initialization* is usually an assignment statement that sets the initial value of the *loop control variable,* which acts as the counter that controls the loop. The *condition* is a Boolean expression that determines whether the loop will repeat. The *iteration* expression defines the amount by which the loop control variable will change each time the loop is repeated. Notice that these three major sections of the loop must be separated by semicolons. The **for** loop will

continue to execute as long as the condition tests true. Once the condition becomes false, the loop will exit, and program execution will resume on the statement following the **for**.

The **for** loop can proceed in a positive or negative fashion, and it can change the loop control variable by any amount. For example, the following program prints the numbers 100 to –100, in decrements of 5:

```
// A negatively running for loop.

using System;

class DecrFor {
  static void Main() {
    int x;

    for(x = 100; x > -100; x -= 5)
      Console.WriteLine(x);
  }
}
```

An important point about **for** loops is that the conditional expression is always tested at the top of the loop. This means that the code inside the loop may not be executed at all if the condition is false to begin with. Here is an example:

```
for(count=10; count < 5; count++)
  x += count; // this statement will not execute
```

This loop will never execute because its control variable, **count**, is greater than 5 when the loop is first entered. This makes the conditional expression, **count < 5**, false from the outset; thus, not even one iteration of the loop will occur.

The **for** loop is most useful when you will be iterating a known number of times. For example, the following program uses two **for** loops to find the prime numbers between 2 and 20. If the number is not prime, then its largest factor is displayed.

```
// Determine if a number is prime. If it is not, then
// display its largest factor.

using System;

class FindPrimes {
  static void Main() {
    int num;
    int i;
    int factor;
    bool isprime;


    for(num = 2; num < 20; num++) {
      isprime = true;
      factor = 0;

      // See if num is evenly divisible.
      for(i=2; i <= num/2; i++) {
        if((num % i) == 0) {
```

```
          // num is evenly divisible. Thus, it is not prime.
          isprime = false;
          factor = i;
        }
      }

      if(isprime)
        Console.WriteLine(num + " is prime.");
      else
        Console.WriteLine("Largest factor of " + num +
                          " is " + factor);
    }
  }
}
```

The output from the program is shown here:

```
2 is prime.
3 is prime.
Largest factor of 4 is 2
5 is prime.
Largest factor of 6 is 3
7 is prime.
Largest factor of 8 is 4
Largest factor of 9 is 3
Largest factor of 10 is 5
11 is prime.
Largest factor of 12 is 6
13 is prime.
Largest factor of 14 is 7
Largest factor of 15 is 5
Largest factor of 16 is 8
17 is prime.
Largest factor of 18 is 9
19 is prime.
```

Some Variations on the for Loop

The **for** is one of the most versatile statements in the C# language because it allows a wide range of variations. They are examined here.

Using Multiple Loop Control Variables

The **for** loop allows you to use two or more variables to control the loop. When using multiple loop control variables, the initialization and increments statements for each variable are separated by commas. Here is an example:

```
// Use commas in a for statement.

using System;

class Comma {
  static void Main() {
    int i, j;
```

```
    for(i=0, j=10; i < j; i++, j--)
      Console.WriteLine("i and j: " + i + " " + j);
  }
}
```

The output from the program is shown here:

```
i and j: 0 10
i and j: 1 9
i and j: 2 8
i and j: 3 7
i and j: 4 6
```

Here, commas separate the two initialization statements and the two iteration expressions. When the loop begins, both **i** and **j** are initialized. Each time the loop repeats, **i** is incremented and **j** is decremented. Multiple loop control variables are often convenient and can simplify certain algorithms. You can have any number of initialization and iteration statements, but in practice, more than two make the **for** loop unwieldy.

Here is a practical use of multiple loop control variables in a **for** statement. This program uses two loop control variables within a single **for** loop to find the largest and smallest factor of a number, in this case 100. Pay special attention to the termination condition. It relies on both loop control variables.

```
// Use commas in a for statement to find the largest and
// smallest factor of a number.

using System;

class Comma {
  static void Main() {
    int i, j;
    int smallest, largest;
    int num;

    num = 100;

    smallest = largest = 1;

    for(i=2, j=num/2; (i <= num/2) & (j >= 2); i++, j--) {

      if((smallest == 1) & ((num % i) == 0))
        smallest = i;

      if((largest == 1) & ((num % j) == 0))
        largest = j;

    }

    Console.WriteLine("Largest factor: " + largest);
    Console.WriteLine("Smallest factor: " + smallest);
  }
}
```

Here is the output from the program:

```
Largest factor: 50
Smallest factor: 2
```

Through the use of two loop control variables, a single **for** loop can find both the smallest and the largest factor of a number. The control variable **i** is used to search for the smallest factor. It is initially set to 2 and incremented until its value exceeds one half of **num**. The control variable **j** is used to search for the largest factor. Its value is initially set to one half the **num** and decremented until it is less than 2. The loop runs until both **i** and **j** are at their termination values. When the loop ends, both factors will have been found.

The Conditional Expression

The conditional expression controlling a **for** loop can be any valid expression that produces a **bool** result. It does not need to involve the loop control variable. For example, in the next program, the **for** loop is controlled by the value of **done**.

```
// Loop condition can be any bool expression.

using System;

class forDemo {
  static void Main() {
    int i, j;
    bool done = false;

    for(i=0, j=100; !done; i++, j--) {

      if(i*i >= j) done = true;

      Console.WriteLine("i, j: " + i + " " + j);
    }
  }
}
```

The output is shown here:

```
i, j: 0 100
i, j: 1 99
i, j: 2 98
i, j: 3 97
i, j: 4 96
i, j: 5 95
i, j: 6 94
i, j: 7 93
i, j: 8 92
i, j: 9 91
i, j: 10 90
```

In this example, the **for** loop iterates until the **bool** variable **done** is true. This variable is set to true inside the loop when **i** squared is greater than or equal to **j**.

Missing Pieces

Some interesting **for** loop variations are created by leaving pieces of the loop definition empty. In C#, it is possible for any or all of the initialization, condition, or iteration portions of the **for** loop to be empty. For example, consider the following program:

```
// Parts of the for can be empty.

using System;

class Empty {
  static void Main() {
    int i;

    for(i = 0; i < 10; ) {
      Console.WriteLine("Pass #" + i);
      i++; // increment loop control var
    }

  }
}
```

Here, the iteration expression of the **for** is empty. Instead, the loop control variable **i** is incremented inside the body of the loop. This means that each time the loop repeats, **i** is tested to see whether it equals 10, but no further action takes place. Of course, since **i** is incremented within the body of the loop, the loop runs normally, displaying the following output:

```
Pass #0
Pass #1
Pass #2
Pass #3
Pass #4
Pass #5
Pass #6
Pass #7
Pass #8
Pass #9
```

In the next example, the initialization portion is also moved out of the **for**:

```
// Move more out of the for loop.

using System;

class Empty2 {
  static void Main() {
    int i;

    i = 0; // move initialization out of loop
    for(; i < 10; ) {
      Console.WriteLine("Pass #" + i);
      i++; // increment loop control var
    }
  }
}
```

In this version, **i** is initialized before the loop begins, rather than as part of the **for**. Normally, you will want to initialize the loop control variable inside the **for**. Placing the initialization outside of the loop is generally done only when the initial value is derived through a complex process that does not lend itself to containment inside the **for** statement.

The Infinite Loop

You can create an *infinite loop* (a loop that never terminates) using the **for** by leaving the conditional expression empty. For example, the following fragment shows the way many C# programmers create an infinite loop:

```
for(;;) // intentionally infinite loop
{
  //...
}
```

This loop will run forever. Although there are some programming tasks, such as operating system command processors, that require an infinite loop, most "infinite loops" are really just loops with special termination requirements. (See "Using **break** to Exit a Loop," later in this chapter.)

Loops with No Body

In C#, the body associated with a **for** loop (or any other loop) can be empty. This is because a *empty statement* is syntactically valid. Bodyless loops are often useful. For example, the following program uses a bodyless loop to sum the numbers 1 through 5:

```
// The body of a loop can be empty.

using System;

class Empty3 {
  static void Main() {
    int i;
    int sum = 0;

    // Sum the numbers through 5.
    for(i = 1; i <= 5; sum += i++) ;

    Console.WriteLine("Sum is " + sum);
  }
}
```

The output from the program is shown here:

```
Sum is 15
```

Notice that the summation process is handled entirely within the **for** statement, and no body is needed. Pay special attention to the iteration expression:

```
sum += i++
```

Don't be intimidated by statements like this. They are common in professionally written C# programs and are easy to understand if you break them down into their parts. In words,

this statement says "add to **sum** the value of **sum** plus **i**, then increment **i**." Thus, it is the same as this sequence of statements:

```
sum = sum + i;
i++;
```

Declaring Loop Control Variables Inside the for Loop

Often the variable that controls a **for** loop is needed only for the purposes of the loop and is not used elsewhere. When this is the case, it is possible to declare the variable inside the initialization portion of the **for**. For example, the following program computes both the summation and the factorial of the numbers 1 through 5. It declares its loop control variable **i** inside the **for**:

```
// Declare loop control variable inside the for.

using System;

class ForVar {
  static void Main() {
    int sum = 0;
    int fact = 1;

    // Compute the factorial of the numbers 1 through 5.
    for(int i = 1; i <= 5; i++) {
      sum += i;  // i is known throughout the loop.
      fact *= i;
    }

    // But, i is not known here.

    Console.WriteLine("Sum is " + sum);
    Console.WriteLine("Factorial is " + fact);
  }
}
```

When you declare a variable inside a **for** loop, there is one important point to remember: The scope of that variable ends when the **for** statement does. (That is, the scope of the variable is limited to the **for** loop.) Outside the **for** loop, the variable will cease to exist. Thus, in the preceding example, **i** is not accessible outside the **for** loop. If you need to use the loop control variable elsewhere in your program, you will not be able to declare it inside the **for** loop.

Before moving on, you might want to experiment with your own variations on the **for** loop. As you will find, it is a fascinating loop.

The while Loop

Another of C#'s loops is the **while**. The general form of the **while** loop is

while(*condition*) *statement*;

where *statement* can be a single statement or a block of statements, and *condition* defines the condition that controls the loop and may be any valid Boolean expression. The statement is performed while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop.

Here is a simple example in which a **while** is used to compute the order of magnitude of an integer:

```
// Compute the order of magnitude of an integer

using System;

class WhileDemo {
  static void Main() {
    int num;
    int mag;

    num = 435679;
    mag = 0;

    Console.WriteLine("Number: " + num);

    while(num > 0) {
      mag++;
      num = num / 10;
    };

    Console.WriteLine("Magnitude: " + mag);
  }
}
```

The output is shown here:

```
Number: 435679
Magnitude: 6
```

The **while** loop works like this: The value of **num** is tested. If **num** is greater than 0, the **mag** counter is incremented, and **num** is divided by 10. As long as the value in **num** is greater than 0, the loop repeats. When **num** is 0, the loop terminates and **mag** contains the order of magnitude of the original value.

As with the **for** loop, the **while** checks the conditional expression at the top of the loop, which means that the loop code may not execute at all. This eliminates the need for performing a separate test before the loop. The following program illustrates this characteristic of the **while** loop. It computes the integer powers of 2 from 0 to 9.

```
// Compute integer powers of 2.

using System;

class Power {
  static void Main() {
    int e;
    int result;

    for(int i=0; i < 10; i++) {
      result = 1;
      e = i;
```

```
      while(e > 0) {
        result *= 2;
        e--;
      }

      Console.WriteLine("2 to the " + i + " power is " + result);
    }
  }
}
```

The output from the program is shown here:

```
2 to the 0 power is 1
2 to the 1 power is 2
2 to the 2 power is 4
2 to the 3 power is 8
2 to the 4 power is 16
2 to the 5 power is 32
2 to the 6 power is 64
2 to the 7 power is 128
2 to the 8 power is 256
2 to the 9 power is 512
```

Notice that the **while** loop executes only when **e** is greater than 0. Thus, when **e** is 0, as it is in the first iteration of the **for** loop, the **while** loop is skipped.

The do-while Loop

The third C# loop is the **do-while**. Unlike the **for** and the **while** loops, in which the condition is tested at the top of the loop, the **do-while** loop checks its condition at the bottom of the loop. This means that a **do-while** loop will always execute at least once. The general form of the **do-while** loop is

```
do {
    statements;
} while(condition);
```

Although the braces are not necessary when only one statement is present, they are often used to improve readability of the **do-while** construct, thus preventing confusion with the **while**. The **do-while** loop executes as long as the conditional expression is true.

The following program uses a **do-while** loop to display the digits of an integer in reverse order:

```
// Display the digits of an integer in reverse order.

using System;

class DoWhileDemo {
  static void Main() {
    int num;
    int nextdigit;

    num = 198;
```

```
    Console.WriteLine("Number: " + num);

    Console.Write("Number in reverse order: ");

    do {
      nextdigit = num % 10;
      Console.Write(nextdigit);
      num = num / 10;
    } while(num > 0);

    Console.WriteLine();
  }
}
```

The output is shown here:

```
Number: 198
Number in reverse order: 891
```

Here is how the loop works: With each iteration, the leftmost digit is obtained by computing the remainder of an integer division by 10. This digit is then displayed. Next, the value in **num** is divided by 10. Since this is an integer division, this results in the leftmost digit being removed. This process repeats until **num** is 0.

The foreach Loop

The **foreach** loop cycles through the elements of a *collection*. A collection is a group of objects. C# defines several types of collections, of which one is an array. The **foreach** loop is examined in Chapter 7, when arrays are discussed.

Using break to Exit a Loop

It is possible to force an immediate exit from a loop, bypassing any code remaining in the body of the loop and the loop's conditional test, by using the **break** statement. When a **break** statement is encountered inside a loop, the loop is terminated, and program control resumes at the next statement following the loop. Here is a simple example:

```
// Using break to exit a loop.

using System;

class BreakDemo {
  static void Main() {

    // Use break to exit this loop.
    for(int i=-10; i <= 10; i++) {
      if(i > 0) break; // terminate loop when i is positive
      Console.Write(i + " ");
    }
    Console.WriteLine("Done");
  }
}
```

This program generates the following output:

```
-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 Done
```

As you can see, although the **for** loop is designed to run from –10 to 10, the **break** statement causes it to terminate early, when **i** becomes positive.

The **break** statement can be used with any of C#'s loops. For example, here is the previous program recoded to use a **do-while** loop:

```
// Using break to exit a do-while loop.

using System;

class BreakDemo2 {
  static void Main() {
    int i;

    i = -10;
    do {
      if(i > 0) break;
      Console.Write(i + " ");
      i++;
    } while(i <= 10);

    Console.WriteLine("Done");
  }
}
```

Here is a more practical example of **break**. This program finds the smallest factor of a number.

```
// Find the smallest factor of a value.

using System;

class FindSmallestFactor {
  static void Main() {
    int factor = 1;
    int num = 1000;

    for(int i=2; i <= num/i; i++) {
      if((num%i) == 0) {
        factor = i;
        break; // stop loop when factor is found
      }
    }
    Console.WriteLine("Smallest factor is " + factor);
  }
}
```

The output is shown here:

```
Smallest factor is 2
```

The **break** stops the **for** loop as soon as a factor is found. The use of **break** in this situation prevents the loop from trying any other values once a factor has been found, thus preventing inefficiency.

When used inside a set of nested loops, the **break** statement will break out of only the innermost loop. For example:

```
// Using break with nested loops.

using System;

class BreakNested {
  static void Main() {

    for(int i=0; i<3; i++) {
      Console.WriteLine("Outer loop count: " + i);
      Console.Write("    Inner loop count: ");

      int t = 0;
      while(t < 100) {
        if(t == 10) break; // terminate loop if t is 10
        Console.Write(t + " ");
        t++;
      }
      Console.WriteLine();
    }
    Console.WriteLine("Loops complete.");
  }
}
```

This program generates the following output:

```
Outer loop count: 0
    Inner loop count: 0 1 2 3 4 5 6 7 8 9
Outer loop count: 1
    Inner loop count: 0 1 2 3 4 5 6 7 8 9
Outer loop count: 2
    Inner loop count: 0 1 2 3 4 5 6 7 8 9
Loops complete.
```

As you can see, the **break** statement in the inner loop causes only the termination of that loop. The outer loop is unaffected.

Here are two other points to remember about **break**: First, more than one **break** statement may appear in a loop. However, be careful. Too many **break** statements have the tendency to destructure your code. Second, the **break** that exits a **switch** statement affects only that **switch** statement and not any enclosing loops.

Using continue

It is possible to force an early iteration of a loop, bypassing the loop's normal control structure. This is accomplished using **continue**. The **continue** statement forces the next iteration of the loop to take place, skipping any code in between. Thus, **continue** is essentially the complement of **break**. For example, the following program uses **continue** to help print the even numbers between 0 and 100.

```
// Use continue.

using System;

class ContDemo {
  static void Main() {

    // Print even numbers between 0 and 100.
    for(int i = 0; i <= 100; i++) {
      if((i%2) != 0) continue; // iterate
      Console.WriteLine(i);
    }
  }
}
```

Only even numbers are printed, because an odd number will cause the loop to iterate early, bypassing the call to **WriteLine( )**.

In **while** and **do-while** loops, a **continue** statement will cause control to go directly to the conditional expression and then continue the looping process. In the case of the **for**, the iteration expression of the loop is evaluated, then the conditional expression is executed, and then the loop continues.

Good uses of **continue** are rare. One reason is that C# provides a rich set of loop statements that fit most applications. However, for those special circumstances in which early iteration is needed, the **continue** statement provides a structured way to accomplish it.

return

The **return** statement causes a method to return. It can also be used to return a value. It is examined in Chapter 6.

The goto

The **goto** is C#'s unconditional jump statement. When encountered, program flow jumps to the location specified by the **goto**. The statement fell out of favor with programmers many years ago because it encouraged the creation of "spaghetti code." However, the **goto** is still occasionally—and sometimes effectively—used. This book will not make a judgment regarding its validity as a form of program control. It should be stated, however, that there are no programming situations that require the use of the **goto** statement—it is not necessary for making the language complete. Rather, **goto** is a convenience that, if used wisely, can be a benefit in certain programming situations. As such, the **goto** is not used in this book outside of this section. The chief concern most programmers have about the **goto** is its tendency to clutter a program and render it nearly unreadable. However, there are times when the use of the **goto** can clarify program flow rather than confuse it.

The **goto** requires a label for operation. A *label* is a valid C# identifier followed by a colon. The label must be in the same method as the **goto** that uses it and within scope. For example, a loop from 1 to 100 could be written using a **goto** and a label, as shown here:

```
x = 1;
loop1:
  x++;
  if(x < 100) goto loop1;
```

The **goto** can also be used to jump to a **case** or **default** statement within a **switch**. Technically, the **case** and **default** statements of a **switch** are labels. Thus, they can be targets of a **goto**. However, the **goto** statement must be executed from within the **switch**. That is, you cannot use the **goto** to jump into a **switch** statement. Here is an example that illustrates **goto** with a **switch**:

```
// Use goto with a switch.

using System;

class SwitchGoto {
  static void Main() {

    for(int i=1; i < 5; i++) {
      switch(i) {
        case 1:
          Console.WriteLine("In case 1");
          goto case 3;
        case 2:
          Console.WriteLine("In case 2");
          goto case 1;
        case 3:
          Console.WriteLine("In case 3");
          goto default;
        default:
          Console.WriteLine("In default");
          break;
      }

      Console.WriteLine();
    }

//    goto case 1; // Error! Can't jump into a switch.
  }
}
```

The output from the program is shown here:

```
In case 1
In case 3
In default

In case 2
In case 1
In case 3
In default

In case 3
In default

In default
```

Inside the **switch**, notice how the **goto** is used to jump to other **case** statements or the **default** statement. Furthermore, notice that the **case** statements do not end with a **break**. Since the **goto** prevents one **case** from falling through to the next, the no fall-through rule is not violated, and there is no need for a **break** statement. As explained, it is not possible to use the **goto** to jump into a **switch**. If you remove the comment symbols from the start of this line

```
//    goto case 1; // Error! Can't jump into a switch.
```

the program will not compile. Frankly, using a **goto** with a **switch** can be useful in some special-case situations, but it is not recommended style in general.

One good use for the **goto** is to exit from a deeply nested routine. Here is a simple example:

```
// Demonstrate the goto.

using System;

class Use_goto {
  static void Main() {
    int i=0, j=0, k=0;

    for(i=0; i < 10; i++) {
      for(j=0; j < 10; j++ ) {
        for(k=0; k < 10; k++) {
          Console.WriteLine("i, j, k: " + i + " " + j + " " + k);
          if(k == 3) goto stop;
        }
      }
    }

stop:
    Console.WriteLine("Stopped! i, j, k: " + i + ", " + j + " " + k);

  }
}
```

The output from the program is shown here:

```
i, j, k: 0 0 0
i, j, k: 0 0 1
i, j, k: 0 0 2
i, j, k: 0 0 3
Stopped! i, j, k: 0, 0 3
```

Eliminating the **goto** would force the use of three **if** and **break** statements. In this case, the **goto** simplifies the code. While this is a contrived example used for illustration, you can probably imagine situations in which a **goto** might be beneficial.

One last point: Although you can jump out of a block (as the preceding example shows), you can't use the **goto** to jump into a block.

This page intentionally left blank

CHAPTER 6

Introducing Classes and Objects

This chapter introduces the class. The class is the foundation of C# because it defines the nature of an object. Furthermore, the class forms the basis for object-oriented programming. Within a class are defined both code and data. Because classes and objects are fundamental to C#, they constitute a large topic, which spans several chapters. This chapter begins the discussion by covering their main features.

Class Fundamentals

We have been using classes since the start of this book. Of course, only extremely simple classes have been used, and we have not taken advantage of the majority of their features. Classes are substantially more powerful than the limited ones presented so far.

Let's begin by reviewing the basics. A *class* is a template that defines the form of an object. It specifies both the data and the code that will operate on that data. C# uses a class specification to construct *objects.* Objects are *instances* of a class. Thus, a class is essentially a set of plans that specify how to build an object. It is important to be clear on one issue: A class is a logical abstraction. It is not until an object of that class has been created that a physical representation of that class exists in memory.

The General Form of a Class

When you define a class, you declare the data that it contains and the code that operates on it. While very simple classes might contain only code or only data, most real-world classes contain both.

In general terms, data is contained in *data members* defined by the class, and code is contained in *function members.* It is important to state at the outset that C# defines several specific flavors of data and function members. For example, data members (also called *fields*) include instance variables and static variables. Function members include methods, constructors, destructors, indexers, events, operators, and properties. For now, we will limit our discussion of the class to its essential elements: instance variables and methods. Later in this chapter constructors and destructors are discussed. The other types of members are described in later chapters.

A class is created by use of the keyword **class**. Here is the general form of a simple **class** definition that contains only instance variables and methods:

```
class classname {
    // declare instance variables
    access type var1;
    access type var2;
    // ...
    access type varN;

    // declare methods
    access ret-type method1(parameters) {
        // body of method
    }
    access ret-type method2(parameters) {
        // body of method
    }
    // ...
    access ret-type methodN(parameters) {
        // body of method
    }
}
```

Notice that each variable and method declaration is preceded with *access*. Here, *access* is an access specifier, such as **public**, which specifies how the member can be accessed. As mentioned in Chapter 2, class members can be private to a class or more accessible. The access specifier determines what type of access is allowed. The access specifier is optional, and if absent, then the member is private to the class. Members with private access can be used only by other members of their class. For the examples in this chapter, all members (except for the **Main( )** method) will be specified as **public**, which means that they can be used by all other code—even code defined outside the class. We will return to the topic of access specifiers in Chapter 8.

NOTE *In addition to an access specifier, the declaration of a class member can also contain one or more type modifiers. These modifiers are discussed later in this book.*

Although there is no syntactic rule that enforces it, a well-designed class should define one and only one logical entity. For example, a class that stores names and telephone numbers will not normally also store information about the stock market, average rainfall, sunspot cycles, or other unrelated information. The point here is that a well-designed class groups logically connected information. Putting unrelated information into the same class will quickly destructure your code.

Up to this point, the classes that we have been using have had only one method: **Main( )**. However, notice that the general form of a class does not specify a **Main( )** method. A **Main( )** method is required only if that class is the starting point for your program.

Define a Class

To illustrate classes, we will be evolving a class that encapsulates information about buildings, such as houses, stores, offices, and so on. This class is called **Building**, and it will store three

items of information about a building: the number of floors, the total area, and the number of occupants.

The first version of **Building** is shown here. It defines three instance variables: **Floors**, **Area**, and **Occupants**. Notice that **Building** does not contain any methods. Thus, it is currently a data-only class. (Subsequent sections will add methods to it.)

```
class Building {
  public int Floors;    // number of floors
  public int Area;      // total square footage of building
  public int Occupants; // number of occupants
}
```

The instance variables defined by **Building** illustrate the way that instance variables are declared in general. The general form for declaring an instance variable is shown here:

access type var-name;

Here, *access* specifies the access; *type* specifies the type of variable; and *var-name* is the variable's name. Thus, aside from the access specifier, you declare an instance variable in the same way that you declare local variables. For **Building**, the variables are preceded by the **public** access modifier. As explained, this allows them to be accessed by code outside of **Building**.

A **class** definition creates a new data type. In this case, the new data type is called **Building**. You will use this name to declare objects of type **Building**. Remember that a **class** declaration is only a type description; it does not create an actual object. Thus, the preceding code does not cause any objects of type **Building** to come into existence.

To actually create a **Building** object, you will use a statement like the following:

```
Building house = new Building(); // create an object of type building
```

After this statement executes, **house** will be an instance of **Building**. Thus, it will have "physical" reality. For the moment, don't worry about the details of this statement.

Each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. Thus, every **Building** object will contain its own copies of the instance variables **Floors**, **Area**, and **Occupants**. To access these variables, you will use the member access operator, which is a period. It is commonly referred to as the *dot operator.* The dot operator links the name of an object with the name of a member. The general form of the dot operator is shown here:

object.member

Thus, the *object* is specified on the left, and the *member* is put on the right. For example, to assign the **Floors** variable of **house** the value 2, use the following statement:

```
house.Floors = 2;
```

In general, you can use the dot operator to access both instance variables and methods.

Here is a complete program that uses the **Building** class:

```
// A program that uses the Building class.

using System;
```

```
class Building {
  public int Floors;    // number of floors
  public int Area;      // total square footage of building
  public int Occupants; // number of occupants
}

// This class declares an object of type Building.
class BuildingDemo {
  static void Main() {
    Building house = new Building(); // create a Building object
    int areaPP; // area per person

    // Assign values to fields in house.
    house.Occupants = 4;
    house.Area = 2500;
    house.Floors = 2;

    // Compute the area per person.
    areaPP = house.Area / house.Occupants;

    Console.WriteLine("house has:\n  " +
                      house.Floors + " floors\n  " +
                      house.Occupants + " occupants\n  " +
                      house.Area + " total area\n  " +
                      areaPP + " area per person");
  }
}
```

This program consists of two classes: **Building** and **BuildingDemo**. Inside **BuildingDemo**, the **Main( )** method creates an instance of **Building** called **house**. Then the code within **Main( )** accesses the instance variables associated with **house**, assigning them values and using those values. It is important to understand that **Building** and **BuildingDemo** are two separate classes. The only relationship they have to each other is that one class creates an instance of the other. Although they are separate classes, code inside **BuildingDemo** can access the members of **Building** because they are declared **public**. If they had not been given the **public** access specifier, their access would have been limited to the **Building** class, and **BuildingDemo** would not have been able to use them.

Assume that you call the preceding file **UseBuilding.cs**. Compiling this program creates a file called **UseBuilding.exe**. Both the **Building** and **BuildingDemo** classes are automatically part of the executable file. The program displays the following output:

```
house has:
  2 floors
  4 occupants
  2500 total area
  625 area per person
```

It is not necessary for the **Building** and the **BuildingDemo** classes to actually be in the same source file. You could put each class in its own file, called **Building.cs** and **BuildingDemo.cs**, for example. Just tell the C# compiler to compile both files and link

them together. For example, you could use this command line to compile the program if you split it into two pieces as just described:

```
csc Building.cs BuildingDemo.cs
```

If you are using the Visual C++ IDE, you will need to add both files to your project and then build.

Before moving on, let's review a fundamental principle: Each object has its own copies of the instance variables defined by its class. Thus, the contents of the variables in one object can differ from the contents of the variables in another. There is no connection between the two objects except for the fact that they are both objects of the same type. For example, if you have two **Building** objects, each has its own copy of **Floors**, **Area**, and **Occupants**, and the contents of these can (and often will) differ between the two objects. The following program demonstrates this fact:

```
// This program creates two Building objects.

using System;

class Building {
  public int Floors;    // number of floors
  public int Area;      // total square footage of building
  public int Occupants; // number of occupants
}

// This class declares two objects of type Building.
class BuildingDemo {
  static void Main() {
    Building house = new Building();
    Building office = new Building();

    int areaPP; // area per person

    // Assign values to fields in house.
    house.Occupants = 4;
    house.Area = 2500;
    house.Floors = 2;

    // Assign values to fields in office.
    office.Occupants = 25;
    office.Area = 4200;
    office.Floors = 3;

    // Compute the area per person in house.
    areaPP = house.Area / house.Occupants;

    Console.WriteLine("house has:\n  " +
                      house.Floors + " floors\n  " +
                      house.Occupants + " occupants\n  " +
                      house.Area + " total area\n  " +
                      areaPP + " area per person");
```

```
    Console.WriteLine();

    // Compute the area per person in office.
    areaPP = office.Area / office.Occupants;

    Console.WriteLine("office has:\n  " +
                      office.Floors + " floors\n  " +
                      office.Occupants + " occupants\n  " +
                      office.Area + " total area\n  " +
                      areaPP + " area per person");
  }
}
```

The output produced by this program is shown here:

```
house has:
  2 floors
  4 occupants
  2500 total area
  625 area per person

office has:
  3 floors
  25 occupants
  4200 total area
  168 area per person
```

As you can see, **house**'s data is completely separate from the data contained in **office**. Figure 6-1 depicts this situation.

How Objects Are Created

In the preceding programs, the following line was used to declare an object of type **Building**:

```
Building house = new Building();
```

This declaration performs three functions. First, it declares a variable called **house** of the class type **Building**. This variable is not, itself, an object. Instead, it is simply a variable that can *refer to* an object. Second, the declaration creates an actual, physical copy of the object.

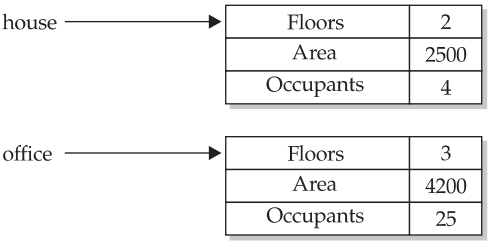


FIGURE 6-1 One object's instance variables are separate from another's.

This is done by using the **new** operator. Finally, it assigns to **house** a reference to that object. Thus, after the line executes, **house** refers to an object of type **Building**.

The **new** operator dynamically allocates (that is, allocates at runtime) memory for an object and returns a reference to it. This reference is then stored in a variable. Thus, in C#, all class objects must be dynamically allocated.

As you might expect, it is possible to separate the declaration of **house** from the creation of the object to which it will refer, as shown here:

```
Building house; // declare reference to object
house = new Building(); // allocate a Building object
```

The first line declares **house** as a reference to an object of type **Building**. Thus, **house** is a variable that can refer to an object, but it is not an object, itself. The next line creates a new **Building** object and assigns a reference to it to **house**. Now, **house** is linked with an object.

The fact that class objects are accessed through a reference explains why classes are called *reference types.* The key difference between value types and reference types is what a variable of each type means. For a value type variable, the variable, itself, contains the value. For example, given

```
int x;
x = 10;
```

x contains the value 10 because **x** is a variable of type **int**, which is a value type. However, in the case of

```
Building house = new Building();
```

house does not, itself, contain the object. Instead, it contains a reference to the object.

Reference Variables and Assignment

In an assignment operation, reference variables act differently than do variables of a value type, such as **int**. When you assign one value type variable to another, the situation is straightforward. The variable on the left receives a *copy* of the *value* of the variable on the right. When you assign one object reference variable to another, the situation is a bit more complicated because the assignment causes the reference variable on the left to refer to the same object to which the reference variable on the right refers. The object, itself, is not copied. The effect of this difference can cause some counterintuitive results. For example, consider the following fragment:

```
Building house1 = new Building();
Building house2 = house1;
```

At first glance, it is easy to think that **house1** and **house2** refer to separate and distinct objects, but this is not the case. Instead, **house1** and **house2** will both refer to the *same* object. The assignment of **house1** to **house2** simply makes **house2** refer to the same object that **house1** does. Thus, the object can be acted upon by either **house1** or **house2**. For example, after the assignment

```
house1.Area = 2600;
```

executes, both of these **WriteLine( )** statements

```
Console.WriteLine(house1.Area);
Console.WriteLine(house2.Area);
```

display the same value: 2600.

Although **house1** and **house2** both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to **house2** simply changes what object **house2** refers to. For example:

```
Building house1 = new Building();
Building house2 = house1;
Building house3 = new Building();

house2 = house3; // now house2 and house3 refer to the same object.
```

After this sequence executes, **house2** refers to the same object as **house3**. The object referred to by **house1** is unchanged.

Methods

As explained, instance variables and methods are two of the primary constituents of classes. So far, the **Building** class contains data, but no methods. Although data-only classes are perfectly valid, most classes will have methods. *Methods* are subroutines that manipulate the data defined by the class and, in many cases, provide access to that data. Typically, other parts of your program will interact with a class through its methods.

A method contains one or more statements. In well-written C# code, each method performs only one task. Each method has a name, and it is this name that is used to call the method. In general, you can name a method using any valid identifier that you please. However, remember that **Main( )** is reserved for the method that begins execution of your program. Also, don't use C#'s keywords for method names.

When denoting methods in text, this book has used and will continue to use a convention that has become common when writing about C#. A method will have parentheses after its name. For example, if a method's name is **GetVal**, then it will be written **GetVal( )** when its name is used in a sentence. This notation will help you distinguish variable names from method names in this book.

The general form of a method is shown here:

```
access ret-type name(parameter-list) {
   // body of method
}
```

Here, *access* is an access modifier that governs what other parts of your program can call the method. As explained earlier, the access modifier is optional. If not present, then the method is private to the class in which it is declared. For now, we will declare methods as **public** so that they can be called by any other code in the program. The *ret-type* specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be **void**. The name of the method is specified by *name.* This can be any legal identifier other than those that would cause conflicts within the current declaration space. The *parameter-list* is a sequence of type

and identifier pairs separated by commas. Parameters are variables that receive the value of the *arguments* passed to the method when it is called. If the method has no parameters, then the parameter list will be empty.

Add a Method to the Building Class

As just explained, the methods of a class typically manipulate and provide access to the data of the class. With this in mind, recall that **Main( )** in the preceding examples computed the area-per-person by dividing the total area by the number of occupants. Although technically correct, this is not the best way to handle this computation. The calculation of area-per-person is something that is best handled by the **Building** class, itself. The reason for this conclusion is easy to understand: The area-per-person of a building is dependent upon the values in the **Area** and **Occupants** fields, which are encapsulated by **Building**. Thus, it is possible for the **Building** class to perform this calculation on its own. Furthermore, by adding this calculation to **Building**, you prevent each program that uses **Building** from having to perform this calculation manually. This prevents the unnecessary duplication of code. Finally, by adding a method to **Building** that computes the area-per-person, you are enhancing its object-oriented structure by encapsulating the quantities that relate directly to a building inside **Building**.

To add a method to **Building**, specify it within **Building**'s declaration. For example, the following version of **Building** contains a method called **AreaPerPerson( )** that displays the area-per-person for a building:

```
// Add a method to Building.

using System;

class Building {
  public int Floors;    // number of floors
  public int Area;      // total square footage of building
  public int Occupants; // number of occupants

  // Display the area per person.
  public void AreaPerPerson() {
    Console.WriteLine("  " + Area / Occupants + " area per person");
  }
}

// Use the AreaPerPerson() method.
class BuildingDemo {
  static void Main() {
    Building house = new Building();
    Building office = new Building();


    // Assign values to fields in house.
    house.Occupants = 4;
    house.Area = 2500;
    house.Floors = 2;

    // Assign values to fields in office.
    office.Occupants = 25;
```

```
    office.Area = 4200;
    office.Floors = 3;


    Console.WriteLine("house has:\n  " +
                      house.Floors + " floors\n  " +
                      house.Occupants + " occupants\n  " +
                      house.Area + " total area");
    house.AreaPerPerson();

    Console.WriteLine();

    Console.WriteLine("office has:\n  " +
                      office.Floors + " floors\n  " +
                      office.Occupants + " occupants\n  " +
                      office.Area + " total area");
    office.AreaPerPerson();
  }
}
```

This program generates the following output, which is the same as before:

```
house has:
  2 floors
  4 occupants
  2500 total area
  625 area per person

office has:
  3 floors
  25 occupants
  4200 total area
  168 area per person
```

Let's look at the key elements of this program, beginning with the **AreaPerPerson( )** method, itself. The first line of **AreaPerPerson( )** is

```
public void AreaPerPerson() {
```

This line declares a method called **AreaPerPerson** that has no parameters. It is specified as **public**, so it can be used by all other parts of the program. Its return type is **void**. Thus, **AreaPerPerson( )** does not return a value to the caller. The line ends with the opening curly brace of the method body.

The body of **AreaPerPerson( )** consists solely of this statement:

```
Console.WriteLine("  " + Area / Occupants + " area per person");
```

This statement displays the area-per-person of a building by dividing **Area** by **Occupants**. Since each object of type **Building** has its own copy of **Area** and **Occupants**, when **AreaPerPerson( )** is called, the computation uses the calling object's copies of those variables.

The **AreaPerPerson( )** method ends when its closing curly brace is encountered. This causes program control to transfer back to the caller.

Next, look closely at this line of code from inside **Main( )**:

```
house.AreaPerPerson();
```

This statement invokes the **AreaPerPerson( )** method on **house**. That is, it calls **AreaPerPerson( )** relative to the object referred to by **house**, by use of the dot operator. When a method is called, program control is transferred to the method. When the method terminates, control is transferred back to the caller, and execution resumes with the line of code following the call.

In this case, the call to **house.AreaPerPerson( )** displays the area-per-person of the building defined by **house**. In similar fashion, the call to **office.AreaPerPerson( )** displays the area-per-person of the building defined by **office**. Each time **AreaPerPerson( )** is invoked, it displays the area-per-person for the specified object.

There is something very important to notice inside the **AreaPerPerson( )** method: The instance variables **Area** and **Occupants** are referred to directly, without use of the dot operator. When a method uses an instance variable that is defined by its class, it does so directly, without explicit reference to an object and without use of the dot operator. This is easy to understand if you think about it. A method is always invoked relative to some object of its class. Once this invocation has occurred, the object is known. Thus, within a method, there is no need to specify the object a second time. This means that **Area** and **Occupants** inside **AreaPerPerson( )** implicitly refer to the copies of those variables found in the object that invokes **AreaPerPerson( )**.

NOTE *As a point of interest, in the* **AreaPerPerson( )** *method,* **Occupants** *must not equal zero (which it won't for all of the examples in this chapter). If* **Occupants** *were zero, then a division-by-zero error would occur. In Chapter 13, you will learn about exceptions, which are C#'s approach to handling errors, and see how to watch for errors that can occur at runtime.*

Return from a Method

In general, there are two conditions that cause a method to return. The first, as the **AreaPerPerson( )** method in the preceding example shows, is when the method's closing curly brace is encountered. The second is when a **return** statement is executed. There are two forms of **return**: one for use in **void** methods (those that do not return a value) and one for returning values. The first form is examined here. The next section explains how to return values.

In a **void** method, you can cause the immediate termination of a method by using this form of **return**:

```
return ;
```

When this statement executes, program control returns to the caller, skipping any remaining code in the method. For example, consider this method:

```
public void MyMeth() {
  int i;

  for(i=0; i<10; i++) {
    if(i == 5) return; // stop at 5
    Console.WriteLine();
  }
}
```

Here, the **for** loop will only run from 0 to 5, because once **i** equals 5, the method returns.

It is permissible to have multiple **return** statements in a method, especially when there are two or more routes out of it. For example,

```
public void MyMeth() {
  // ...
  if(done) return;
  // ...
  if(error) return;
}
```

Here, the method returns if it is done or if an error occurs. Be careful, however. Having too many exit points in a method can destructure your code, so avoid using them casually.

To review: A **void** method can return in one of two ways—its closing curly brace is reached, or a **return** statement is executed.

Return a Value

Although methods with a return type of **void** are not rare, most methods will return a value. In fact, the ability to return a value is one of a method's most useful features. You have already seen an example of a return value when we used the **Math.Sqrt( )** function in Chapter 3 to obtain a square root.

Return values are used for a variety of purposes in programming. In some cases, such as with **Math.Sqrt( )**, the return value contains the outcome of some calculation. In other cases, the return value may simply indicate success or failure. In still others, it may contain a status code. Whatever the purpose, using method return values is an integral part of C# programming.

Methods return a value to the calling routine using this form of **return**:

return *value*;

Here, *value* is the value returned.

You can use a return value to improve the implementation of **AreaPerPerson( )**. Instead of displaying the area-per-person, a better approach is to have **AreaPerPerson( )** return this value. Among the advantages to this approach is that you can use the value for other calculations. The following example modifies **AreaPerPerson( )** to return the area-per-person rather than displaying it:

```
// Return a value from AreaPerPerson().

using System;

class Building {
  public int Floors;    // number of floors
  public int Area;      // total square footage of building
  public int Occupants; // number of occupants

  // Return the area per person.
  public int AreaPerPerson() {
    return Area / Occupants;
  }
}
```

```
// Use the return value from AreaPerPerson().
class BuildingDemo {
  static void Main() {
    Building house = new Building();
    Building office = new Building();
    int areaPP; // area per person

    // Assign values to fields in house.
    house.Occupants = 4;
    house.Area = 2500;
    house.Floors = 2;

    // Assign values to fields in office.
    office.Occupants = 25;
    office.Area = 4200;
    office.Floors = 3;

    // Obtain area per person for house.
    areaPP = house.AreaPerPerson();

    Console.WriteLine("house has:\n  " +
                      house.Floors + " floors\n  " +
                      house.Occupants + " occupants\n  " +
                      house.Area + " total area\n  " +
                      areaPP + " area per person");


    Console.WriteLine();

    // Obtain area per person for office.
    areaPP = office.AreaPerPerson();

    Console.WriteLine("office has:\n  " +
                      office.Floors + " floors\n  " +
                      office.Occupants + " occupants\n  " +
                      office.Area + " total area\n  " +
                      areaPP + " area per person");
  }
}
```

The output is the same as shown earlier.

In the program, notice that when **AreaPerPerson( )** is called, it is put on the right side of an assignment statement. On the left is a variable that will receive the value returned by **AreaPerPerson( )**. Thus, after

```
areaPP = house.AreaPerPerson();
```

executes, the area-per-person of the **house** object is stored in **areaPP**.

Notice that **AreaPerPerson( )** now has a return type of **int**. This means that it will return an integer value to the caller. The return type of a method is important because the type of data returned by a method must be compatible with the return type specified by the method. Thus, if you want a method to return data of type **double**, then its return type must be type **double**.

Although the preceding program is correct, it is not written as efficiently as it could be. Specifically, there is no need for the **areaPP** variable. A call to **AreaPerPerson( )** can be used in the **WriteLine( )** statement directly, as shown here:

```
Console.WriteLine("house has:\n  " +
                  house.Floors + " floors\n  " +
                  house.Occupants + " occupants\n  " +
                  house.Area + " total area\n  " +
                  house.AreaPerPerson() + " area per person");
```

In this case, when **WriteLine( )** is executed, **house.AreaPerPerson( )** is called automatically, and its value will be passed to **WriteLine( )**. Furthermore, you can use a call to **AreaPerPerson( )** whenever the area-per-person of a **Building** object is needed. For example, this statement compares the per-person areas of two buildings:

```
if(b1.AreaPerPerson() > b2.AreaPerPerson())
  Console.WriteLine("b1 has more space for each person");
```

Use Parameters

It is possible to pass one or more values to a method when the method is called. A value passed to a method is called an *argument.* Inside the method, the variable that receives the argument is called a *formal parameter,* or just *parameter,* for short. Parameters are declared inside the parentheses that follow the method's name. The parameter declaration syntax is the same as that used for variables. The scope of a parameter is the body of its method. Aside from its special task of receiving an argument, it acts like any other local variable.

Here is a simple example that uses a parameter. Inside the **ChkNum** class, the method **IsPrime( )** returns **true** if the value that it is passed is prime. It returns **false** otherwise. Therefore, **IsPrime( )** has a return type of **bool**.

```
// A simple example that uses a parameter.

using System;

class ChkNum {
  // Return true if x is prime.
  public bool IsPrime(int x) {
    if(x <= 1) return false;

    for(int i=2; i <= x/i; i++)
      if((x %i) == 0) return false;

    return true;
  }
}

class ParmDemo {
  static void Main() {
    ChkNum ob = new ChkNum();

    for(int i=2; i < 10; i++)
      if(ob.IsPrime(i)) Console.WriteLine(i + " is prime.");
```

```
      else Console.WriteLine(i + " is not prime.");
  }
}
```

Here is the output produced by the program:

```
2 is prime.
3 is prime.
4 is not prime.
5 is prime.
6 is not prime.
7 is prime.
8 is not prime.
9 is not prime.
```

In the program, **IsPrime( )** is called nine times, and each time a different value is passed. Let's look at this process closely. First, notice how **IsPrime( )** is called. The argument is specified between the parentheses. When **IsPrime( )** is called the first time, it is passed value 1. Thus, when **IsPrime( )** begins executing, the parameter **x** receives the value 1. In the second call, 2 is the argument, and **x** then has the value 2. In the third call, the argument is 3, which is the value that **x** receives, and so on. The point is that the value passed as an argument when **IsPrime( )** is called is the value received by its parameter, **x**.

A method can have more than one parameter. Simply declare each parameter, separating one from the next with a comma. For example, here the **ChkNum** class is expanded by adding a method called **LeastComFactor( )**, which returns the smallest factor that its two arguments have in common. In other words, it returns the smallest whole number value that can evenly divide both arguments.

```
// Add a method that takes two arguments.

using System;

class ChkNum {
  // Return true if x is prime.
  public bool IsPrime(int x) {
    if(x <= 1) return false;

    for(int i=2; i <= x/i; i++)
      if((x %i) == 0) return false;

    return true;
  }

  // Return the least common factor.
  public int LeastComFactor(int a, int b) {
    int max;

    if(IsPrime(a) || IsPrime(b)) return 1;

    max = a < b ? a : b;

    for(int i=2; i <= max/2; i++)
      if(((a%i) == 0) && ((b%i) == 0)) return i;
```

```
    return 1;
  }
}

class ParmDemo {
  static void Main() {
    ChkNum ob = new ChkNum();
    int a, b;

    for(int i=2; i < 10; i++)
      if(ob.IsPrime(i)) Console.WriteLine(i + " is prime.");
      else Console.WriteLine(i + " is not prime.");

    a = 7;
    b = 8;
    Console.WriteLine("Least common factor for " +
                      a + " and " + b + " is " +
                      ob.LeastComFactor(a, b));

    a = 100;
    b = 8;
    Console.WriteLine("Least common factor for " +
                      a + " and " + b + " is " +
                      ob.LeastComFactor(a, b));

    a = 100;
    b = 75;
    Console.WriteLine("Least common factor for " +
                      a + " and " + b + " is " +
                      ob.LeastComFactor(a, b));

  }
}
```

Notice that when **LeastComFactor( )** is called, the arguments are also separated by commas. The output from the program is shown here:

```
2 is prime.
3 is prime.
4 is not prime.
5 is prime.
6 is not prime.
7 is prime.
8 is not prime.
9 is not prime.
Least common factor for 7 and 8 is 1
Least common factor for 100 and 8 is 2
Least common factor for 100 and 75 is 5
```

When using multiple parameters, each parameter specifies its own type, which can differ from the others. For example, this is perfectly valid:

```
int MyMeth(int a, double b, float c) {
 // ...
```

Add a Parameterized Method to Building

You can use a parameterized method to add a new feature to the **Building** class: the ability to compute the maximum number of occupants for a building assuming that each occupant must have a certain minimal space. This new method is called **MaxOccupant( )**. It is shown here:

```
// Return the maximum number of occupants if each
// is to have at least the specified minimum area.
public int MaxOccupant(int minArea) {
  return Area / minArea;
}
```

When **MaxOccupant( )** is called, the parameter **minArea** receives the minimum space needed for each occupant. The method divides the total area of the building by this value and returns the result.

The entire **Building** class that includes **MaxOccupant( )** is shown here:

```
/*
   Add a parameterized method that computes the
   maximum number of people that can occupy a
   building assuming each needs a specified
   minimum space.
*/

using System;

class Building {
  public int Floors;    // number of floors
  public int Area;      // total square footage of building
  public int Occupants; // number of occupants

  // Return the area per person.
  public int AreaPerPerson() {
    return Area / Occupants;
  }

  // Return the maximum number of occupants if each
  // is to have at least the specified minimum area.
  public int MaxOccupant(int minArea) {
    return Area / minArea;
  }
}

// Use MaxOccupant().
class BuildingDemo {
  static void Main() {
    Building house = new Building();
    Building office = new Building();

    // Assign values to fields in house.
    house.Occupants = 4;
    house.Area = 2500;
    house.Floors = 2;
```

```
    // Assign values to fields in office.
    office.Occupants = 25;
    office.Area = 4200;
    office.Floors = 3;

    Console.WriteLine("Maximum occupants for house if each has " +
                      300 + " square feet: " +
                      house.MaxOccupant(300));

    Console.WriteLine("Maximum occupants for office if each has " +
                      300 + " square feet: " +
                      office.MaxOccupant(300));
  }
}
```

The output from the program is shown here:

```
Maximum occupants for house if each has 300 square feet: 8
Maximum occupants for office if each has 300 square feet: 14
```

Avoiding Unreachable Code

When creating methods, you should avoid causing a situation in which a portion of code cannot, under any circumstances, be executed. This is called *unreachable code*, and it is considered incorrect in C#. The compiler will issue a warning message if you create a method that contains unreachable code. For example:

```
public void MyMeth() {
  char a, b;

  // ...

  if(a==b) {
    Console.WriteLine("equal");
    return;
  } else {
    Console.WriteLine("not equal");
    return;
  }
  Console.WriteLine("this is unreachable");
}
```

Here, the method **MyMeth( )** will always return before the final **WriteLine( )** statement is executed. If you try to compile this method, you will receive a warning. In general, unreachable code constitutes a mistake on your part, so it is a good idea to take unreachable code warnings seriously.

Constructors

In the preceding examples, the instance variables of each **Building** object had to be set manually using a sequence of statements, such as

```
house.Occupants = 4;
house.Area = 2500;
house.Floors = 2;
```

An approach like this would never be used in professionally written C# code. Aside from this approach being error prone (you might forget to set one of the fields), there is simply a better way to accomplish this task: the constructor.

A *constructor* initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type. The general form of a constructor is shown here:

access class-name(*param-list*) {
  // constructor code
}

Typically, you will use a constructor to give initial values to the instance variables defined by the class or to perform any other startup procedures required to create a fully formed object. Also, usually, *access* is **public** because constructors are normally called from outside their class. The *param-list* can be empty, or it can specify one or more parameters.

All classes have constructors, whether you define one or not, because C# automatically provides a default constructor that causes all member variables to be initialized to their default values. For most value types, the default value is zero. For **bool**, the default is **false**. For reference types, the default is null. However, once you define your own constructor, the default constructor is no longer used.

Here is a simple example that uses a constructor:

```
// A simple constructor.

using System;

class MyClass {
  public int x;

  public MyClass() {
    x = 10;
  }
}

class ConsDemo {
  static void Main() {
    MyClass t1 = new MyClass();
    MyClass t2 = new MyClass();

    Console.WriteLine(t1.x + " " + t2.x);
  }
}
```

In this example, the constructor for **MyClass** is

```
public MyClass() {
  x = 10;
}
```

Notice that the constructor is specified as **public**. This is because the constructor will be called from code defined outside of its class. This constructor assigns the instance variable **x** of **MyClass** the value 10. This constructor is called by **new** when an object is created. For example, in the line

```
MyClass t1 = new MyClass();
```

the constructor **MyClass( )** is called on the **t1** object, giving **t1.x** the value 10. The same is true for **t2**. After construction, **t2.x** has the value 10. Thus, the output from the program is

```
10 10
```

Parameterized Constructors

In the preceding example, a parameterless constructor was used. While this is fine for some situations, most often you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way they are added to a method: just declare them inside the parentheses after the constructor's name. For example, here **MyClass** is given a parameterized constructor:

```
// A parameterized constructor.

using System;

class MyClass {
  public int x;

  public MyClass(int i) {
    x = i;
  }
}

class ParmConsDemo {
  static void Main() {
    MyClass t1 = new MyClass(10);
    MyClass t2 = new MyClass(88);

    Console.WriteLine(t1.x + " " + t2.x);
  }
}
```

The output from this program is shown here:

```
10 88
```

In this version of the program, the **MyClass( )** constructor defines one parameter called **i**, which is used to initialize the instance variable, **x**. Thus, when the line

```
MyClass t1 = new MyClass(10);
```

executes, the value 10 is passed to **i**, which is then assigned to **x**.

Add a Constructor to the Building Class

We can improve the **Building** class by adding a constructor that automatically initializes the **Floors**, **Area**, and **Occupants** fields when an object is constructed. Pay special attention to how **Building** objects are created.

```
// Add a constructor to Building.

using System;

class Building {
  public int Floors;    // number of floors
  public int Area;      // total square footage of building
  public int Occupants; // number of occupants

  // A parameterized constructor for Building.
  public Building(int f, int a, int o) {
    Floors = f;
    Area = a;
    Occupants = o;
  }

  // Display the area per person.
  public int AreaPerPerson() {
    return Area / Occupants;
  }

  // Return the maximum number of occupants if each
  // is to have at least the specified minimum area.
  public int MaxOccupant(int minArea) {
    return Area / minArea;
  }
}

// Use the parameterized Building constructor.
class BuildingDemo {
  static void Main() {
    Building house = new Building(2, 2500, 4);
    Building office = new Building(3, 4200, 25);

    Console.WriteLine("Maximum occupants for house if each has " +
                      300 + " square feet: " +
                      house.MaxOccupant(300));

    Console.WriteLine("Maximum occupants for office if each has " +
                      300 + " square feet: " +
                      office.MaxOccupant(300));
  }
}
```

The output from this program is the same as for the previous version.

Both **house** and **office** were initialized by the **Building( )** constructor when they were created. Each object is initialized as specified in the parameters to its constructor. For example, in the following line,

```
Building house = new Building(2, 2500, 4);
```

the values 2, 2500, and 4 are passed to the **Building( )** constructor when **new** creates the object. Thus, **house**'s copy of **Floors**, **Area**, and **Occupants** will contain the values 2, 2500, and 4, respectively.

The new Operator Revisited

Now that you know more about classes and their constructors, let's take a closer look at the **new** operator. As it relates to classes, the **new** operator has this general form:

new *class-name*(*arg-list*)

Here, *class-name* is the name of the class that is being instantiated. The class name followed by parentheses specifies the constructor for the class. If a class does not define its own constructor, **new** will use the default constructor supplied by C#. Thus, **new** can be used to create an object of any class type.

Since memory is finite, it is possible that **new** will not be able to allocate memory for an object because insufficient memory exists. If this happens, a runtime exception will occur. (You will learn how to handle exceptions in Chapter 13.) For the sample programs in this book, you won't need to worry about running out of memory, but you may need to consider this possibility in real-world programs that you write.

Using new with Value Types

At this point, you might be asking why you don't need to use **new** for variables of the value types, such as **int** or **float**? In C#, a variable of a value type contains its own value. Memory to hold this value is automatically provided when the program is run. Thus, there is no need to explicitly allocate this memory using **new**. Conversely, a reference variable stores a reference to an object. The memory to hold this object must be allocated dynamically, during execution.

Not making the fundamental types, such **int** or **char**, into reference types greatly improves your program's performance. When using a reference type, there is a layer of indirection that adds overhead to each object access. This layer of indirection is avoided by a value type.

As a point of interest, it is permitted to use **new** with the value types, as shown here:

```
int i = new int();
```

Doing so invokes the default constructor for type **int**, which initializes **i** to zero. For example:

```
// Use new with a value type.

using System;

class newValue {
  static void Main() {
    int i = new int(); // initialize i to zero
```

```
    Console.WriteLine("The value of i is: " + i);
  }
}
```

The output from this program is

```
The value of i is: 0
```

As the output verifies, **i** is initialized to zero. Remember, without the use of **new**, **i** would be uninitialized, and it would cause an error to attempt to use it in the **WriteLine( )** statement without explicitly giving it a value first.

In general, invoking **new** for a value type invokes the default constructor for that type. It does not, however, dynamically allocate memory. Frankly, most programmers do not use **new** with the value types.

Garbage Collection and Destructors

As you have seen, objects are dynamically allocated from a pool of free memory by using the **new** operator. Of course, memory is not infinite, and the free memory can be exhausted. Thus, it is possible for **new** to fail because there is insufficient free memory to create the desired object. For this reason, one of the key components of any dynamic allocation scheme is the recovery of free memory from unused objects, making that memory available for subsequent reallocation. In many programming languages, the release of previously allocated memory is handled manually. For example, in C++, the **delete** operator is used to free memory that was allocated. However, C# uses a different, more trouble-free approach: *garbage collection*.

C#'s garbage collection system reclaims objects automatically—occurring transparently, behind the scenes, without any programmer intervention. It works like this: When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object is eventually released and collected. This recycled memory can then be used for a subsequent allocation.

Garbage collection occurs only sporadically during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Thus, you can't know, or make assumptions about, precisely when garbage collection will take place.

Destructors

It is possible to define a method that will be called just prior to an object's final destruction by the garbage collector. This method is called a *destructor*, and it can be used in some highly specialized situations to ensure that an object terminates cleanly. For example, you might use a destructor to ensure that a system resource owned by an object is released. It must be stated at the outset that destructors are a very advanced feature that are applicable only to certain rare cases. They are not normally needed. They are briefly described here for completeness.

Destructors have this general form:

```
~class-name( ) {
  // destruction code
}
```

Here, *class-name* is the name of the class. Thus, a destructor is declared like a constructor except that it is preceded with a ~ (tilde). Notice it has no return type and takes no arguments.

To add a destructor to a class, you simply include it as a member. It is called whenever an object of its class is about to be recycled. Inside the destructor, you will specify those actions that must be performed before an object is destroyed.

It is important to understand that the destructor is called just prior to garbage collection. It is not called when a variable containing a reference to an object goes out of scope, for example. (This differs from destructors in C++, which *are* called when an object goes out of scope.) This means that you cannot know precisely when a destructor will be executed. Furthermore, it is possible for your program to end before garbage collection occurs, so a destructor might not get called at all.

The following program demonstrates a destructor. It works by creating and destroying a large number of objects. During this process, at some point the garbage collector will be activated, and the destructors for the objects will be called.

```
// Demonstrate a destructor.

using System;

class Destruct {
  public int x;

  public Destruct(int i) {
    x = i;
  }

  // Called when object is recycled.
  ~Destruct() {
    Console.WriteLine("Destructing " + x);
  }

  // Generates an object that is immediately destroyed.
  public void Generator(int i) {
    Destruct o = new Destruct(i);
  }

}

class DestructDemo {
  static void Main() {
    int count;

    Destruct ob = new Destruct(0);

    /* Now, generate a large number of objects. At
       some point, garbage collection will occur.
       Note: You might need to increase the number
       of objects generated in order to force
       garbage collection. */

    for(count=1; count < 100000; count++)
```

```
      ob.Generator(count);

    Console.WriteLine("Done");
  }
}
```

Here is how the program works. The constructor sets the instance variable **x** to a known value. In this example, **x** is used as an object ID. The destructor displays the value of **x** when an object is recycled. Of special interest is **Generator( )**. This method creates and then promptly destroys a **Destruct** object. The **DestructDemo** class creates an initial **Destruct** object called **ob**. Then using **ob**, it creates 100,000 objects by calling **Generator( )** on **ob**. This has the net effect of creating and destroying 100,000 objects. At various points in the middle of this process, garbage collection will take place. Precisely how often or when is dependent upon several factors, such as the initial amount of free memory, the operating system, and so on. However, at some point, you will start to see the messages generated by the destructor. If you don't see the messages prior to program termination (that is, before you see the "Done" message), try increasing the number of objects being generated by upping the count in the **for** loop.

One important point: The call to **WriteLine( )** inside **~Destruct( )** is purely for the sake of illustration in this rather contrived example. Normally, a destructor should act only on the instance variables defined by its class.

Because of the nondeterministic way in which destructors are called, they should not be used to perform actions that must occur at a specific point in your program. One other point: It is possible to request garbage collection. This is described in Part II, when C#'s class library is discussed. However, manually initiating garbage collection is not recommended for most circumstances, because it can lead to inefficiencies. Also, because of the way the garbage collector works, even if you explicitly request garbage collection, there is no way to know precisely when a specific object will be recycled.

The this Keyword

Before concluding this chapter, it is necessary to introduce **this**. When a method is called, it is automatically passed a reference to the invoking object (that is, the object on which the method is called). This reference is called **this**. Therefore, **this** refers to the object on which the method is acting. To understand **this**, first consider a program that creates a class called **Rect** that encapsulates the width and height of a rectangle and that includes a method called **Area( )** that returns its area.

```
using System;

class Rect {
  public int Width;
  public int Height;

  public Rect(int w, int h) {
    Width = w;
    Height = h;
  }

  public int Area() {
```

```
    return Width * Height;
  }
}

class UseRect {
  static void Main() {
    Rect r1 = new Rect(4, 5);
    Rect r2 = new Rect(7, 9);

    Console.WriteLine("Area of r1: " + r1.Area());

    Console.WriteLine("Area of r2: " + r2.Area());
  }
}
```

As you know, within a method, the other members of a class can be accessed directly, without any object or class qualification. Thus, inside **Area( )**, the statement

```
return Width * Height;
```

means that the copies of **Width** and **Height** associated with the invoking object will be multiplied together and the result returned. However, the same statement can also be written like this:

```
return this.Width * this.Height;
```

Here, **this** refers to the object on which **Area( )** was called. Thus, **this.Width** refers to that object's copy of **Width**, and **this.Height** refers to that object's copy of **Height**. For example, if **Area( )** had been invoked on an object called **x**, then **this** in the preceding statement would have been referring to **x**. Writing the statement without using **this** is really just shorthand.

It is also possible to use **this** inside a constructor. In this case, **this** refers to the object that is being constructed. For example, inside **Rect( )**, the statements

```
Width = w;
Height = h;
```

can be written like this:

```
this.Width = w;
this.Height = h;
```

Of course, there is no benefit in doing so in this case.

For the sake of illustration, here is the entire **Rect** class written using the **this** reference:

```
using System;

class Rect {
  public int Width;
  public int Height;

  public Rect(int w, int h) {
```

```
    this.Width = w;
    this.Height = h;
  }

  public int Area() {
    return this.Width * this.Height;
  }
}

class UseRect {
  static void Main() {
    Rect r1 = new Rect(4, 5);
    Rect r2 = new Rect(7, 9);

    Console.WriteLine("Area of r1: " + r1.Area());

    Console.WriteLine("Area of r2: " + r2.Area());

  }
}
```

Actually, no C# programmer would use **this** as just shown because nothing is gained and the standard form is easier. However, **this** has some important uses. For example, the C# syntax permits the name of a parameter or a local variable to be the same as the name of an instance variable. When this happens, the local name *hides* the instance variable. You can gain access to the hidden instance variable by referring to it through **this**. For example, the following is a syntactically valid way to write the **Rect( )** constructor:

```
public Rect(int Width, int Height) {
  this.Width = Width;
  this.Height = Height;
}
```

In this version, the names of the parameters are the same as the names of the instance variables, thus hiding them. However, **this** is used to "uncover" the instance variables.

This page intentionally left blank

CHAPTER 7

Arrays and Strings

This chapter returns to the subject of C#'s data types. It discusses arrays and the **string** type. The **foreach** loop is also examined.

Arrays

An *array* is a collection of variables of the same type that are referred to by a common name. In C#, arrays can have one or more dimensions, although the one-dimensional array is the most common. Arrays are used for a variety of purposes because they offer a convenient means of grouping together related variables. For example, you might use an array to hold a record of the daily high temperature for a month, a list of stock prices, or your collection of programming books.

The principal advantage of an array is that it organizes data in such a way that it can be easily manipulated. For example, if you have an array containing the dividends for a selected group of stocks, it is easy to compute the average income by cycling through the array. Also, arrays organize data in such a way that it can be easily sorted.

Although arrays in C# can be used just like arrays in many other programming languages, they have one special attribute: They are implemented as objects. This fact is one reason that a discussion of arrays was deferred until objects had been introduced. By implementing arrays as objects, several important advantages are gained, not the least of which is that unused arrays can be garbage-collected.

One-Dimensional Arrays

A *one-dimensional array* is a list of related variables. Such lists are common in programming. For example, you might use a one-dimensional array to store the account numbers of the active users on a network. Another array might store the current batting averages for a baseball team.

Because arrays in C# are implemented as objects, two steps are needed to obtain an array for use in your program. First, you must declare a variable that can refer to an array. Second, you must create an instance of the array by use of **new**. Therefore, to declare a one-dimensional array, you will typically use this general form:

type[ ] *array-name* = new *type*[*size*];

Here, *type* declares the *element type* of the array. The element type determines the data type of each element that comprises the array. Notice the square brackets that follow *type.* They indicate that a one-dimensional array is being declared. The number of elements that the array will hold is determined by *size.*

NOTE *If you come from a C or C++ background, pay special attention to the way arrays are declared. Specifically, the square brackets follow the type name, not the array name.*

Here is an example. The following creates an **int** array of ten elements and links it to an array reference variable named **sample**.

```
int[] sample = new int[10];
```

The **sample** variable holds a reference to the memory allocated by **new**. This memory is large enough to hold ten elements of type **int**.

As is the case when creating an instance of a class, it is possible to break the preceding declaration in two. For example:

```
int[] sample;
sample = new int[10];
```

In this case, when **sample** is first created, it refers to no physical object. It is only after the second statement executes that **sample** refers to an array.

An individual element within an array is accessed by use of an index. An *index* describes the position of an element within an array. In C#, all arrays have 0 as the index of their first element. Because **sample** has 10 elements, it has index values of 0 through 9. To index an array, specify the number of the element you want, surrounded by square brackets. Thus, the first element in **sample** is **sample[0]**, and the last element is **sample[9]**. For example, the following program loads **sample** with the numbers 0 through 9:

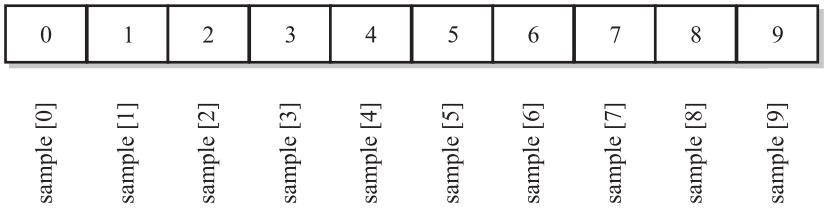
```
// Demonstrate a one-dimensional array.

using System;

class ArrayDemo {
  static void Main() {
    int[] sample = new int[10];
    int i;

    for(i = 0; i < 10; i = i+1)
      sample[i] = i;

    for(i = 0; i < 10; i = i+1)
      Console.WriteLine("sample[" + i + "]: " + sample[i]);
  }
}
```

The output from the program is shown here:

```
sample[0]: 0
sample[1]: 1
sample[2]: 2
```

```
sample[3]: 3
sample[4]: 4
sample[5]: 5
sample[6]: 6
sample[7]: 7
sample[8]: 8
sample[9]: 9
```

Conceptually, the **sample** array looks like this:



Arrays are common in programming because they let you deal easily with large numbers of related variables. For example, the following program finds the average of the set of values stored in the **nums** array by cycling through the array using a **for** loop:

```
// Compute the average of a set of values.

using System;

class Average {
  static void Main() {
    int[] nums = new int[10];
    int avg = 0;

    nums[0] = 99;
    nums[1] = 10;
    nums[2] = 100;
    nums[3] = 18;
    nums[4] = 78;
    nums[5] = 23;
    nums[6] = 63;
    nums[7] = 9;
    nums[8] = 87;
    nums[9] = 49;

    for(int i=0; i < 10; i++)
      avg = avg + nums[i];

    avg = avg / 10;

    Console.WriteLine("Average: " + avg);
  }
}
```

The output from the program is shown here:

```
Average: 53
```

Initializing an Array

In the preceding program, the **nums** array was given values by hand, using ten separate assignment statements. While that is perfectly correct, there is an easier way to accomplish this. Arrays can be initialized when they are created. The general form for initializing a one-dimensional array is shown here:

type[ ] *array-name* = { *val1*, *val2*, *val3*, ..., *valN* };

Here, the initial values are specified by *val1* through *valN*. They are assigned in sequence, left to right, in index order. C# automatically allocates an array large enough to hold the initializers that you specify. There is no need to use the **new** operator explicitly. For example, here is a better way to write the **Average** program:

```
// Compute the average of a set of values.

using System;

class Average {
  static void Main() {
    int[] nums = { 99, 10, 100, 18, 78, 23,
                   63, 9, 87, 49 };
    int avg = 0;

    for(int i=0; i < 10; i++)
      avg = avg + nums[i];

    avg = avg / 10;

    Console.WriteLine("Average: " + avg);
  }
}
```

As a point of interest, although not needed, you can use **new** when initializing an array. For example, this is a proper, but redundant, way to initialize **nums** in the foregoing program:

```
int[] nums = new int[] { 99, 10, 100, 18, 78, 23,
                         63, 9, 87, 49 };
```

Although redundant here, the **new** form of array initialization is useful when you are assigning a new array to an already-existent array reference variable. For example:

```
int[] nums;
nums = new int[] { 99, 10, 100, 18, 78, 23,
                   63, 9, 87, 49 };
```

In this case, **nums** is declared in the first statement and initialized by the second.

One last point: It is permissible to specify the array size explicitly when initializing an array, but the size must agree with the number of initializers. For example, here is another way to initialize **nums**:

```
int[] nums = new int[10] { 99, 10, 100, 18, 78, 23,
                           63, 9, 87, 49 };
```

In this declaration, the size of **nums** is explicitly stated as 10.

Boundaries Are Enforced

Array boundaries are strictly enforced in C#; it is a runtime error to overrun or underrun the ends of an array. If you want to confirm this for yourself, try the following program that purposely overruns an array:

```
// Demonstrate an array overrun.

using System;

class ArrayErr {
  static void Main() {
    int[] sample = new int[10];
    int i;

    // Generate an array overrun.
    for(i = 0; i < 100; i = i+1)
      sample[i] = i;
  }
}
```

As soon as **i** reaches 10, an **IndexOutOfRangeException** is generated and the program is terminated. (See Chapter 13 for a discussion of exceptions and exception handling.)

Multidimensional Arrays

Although the one-dimensional array is the most commonly used array in programming, multidimensional arrays are certainly not rare. A *multidimensional array* is an array that has two or more dimensions, and an individual element is accessed through the combination of two or more indices.

Two-Dimensional Arrays

The simplest form of the multidimensional array is the two-dimensional array. In a two-dimensional array, the location of any specific element is specified by two indices. If you think of a two-dimensional array as a table of information, one index indicates the row, the other indicates the column.

To declare a two-dimensional integer array **table** of size 10, 20, you would write

```
int[,] table = new int[10, 20];
```

Pay careful attention to the declaration. Notice that the two dimensions are separated from each other by a comma. In the first part of the declaration, the syntax

```
[,]
```

indicates that a two-dimensional array reference variable is being created. When memory is actually allocated for the array using **new**, this syntax is used:

```
int[10, 20]
```

This creates a 10×20 array, and again, the comma separates the dimensions.

To access an element in a two-dimensional array, you must specify both indices, separating the two with a comma. For example, to assign the value 10 to location 3, 5 of array **table**, you would use

```
table[3, 5] = 10;
```

Here is a complete example. It loads a two-dimensional array with the numbers 1 through 12 and then displays the contents of the array.

```
// Demonstrate a two-dimensional array.

using System;

class TwoD {
  static void Main() {
    int t, i;
    int[,] table = new int[3, 4];

    for(t=0; t < 3; ++t) {
      for(i=0; i < 4; ++i) {
        table[t,i] = (t*4)+i+1;
        Console.Write(table[t,i] + " ");
      }
      Console.WriteLine();
    }
  }
}
```

In this example, **table[0, 0]** will have the value 1, **table[0, 1]** the value 2, **table[0, 2]** the value 3, and so on. The value of **table[2, 3]** will be 12. Conceptually, the array will look like the one shown in Figure 7-1.

NOTE *If you have previously programmed in C, C++, or Java, be careful when declaring or accessing multidimensional arrays in C#. In these other languages, array dimensions and indices are specified within their own set of brackets. C# separates dimensions using commas.*

Arrays of Three or More Dimensions

C# allows arrays with more than two dimensions. Here is the general form of a multidimensional array declaration:

type[, ...,] *name* = new *type*[*size1*, *size2*, ..., *sizeN*];

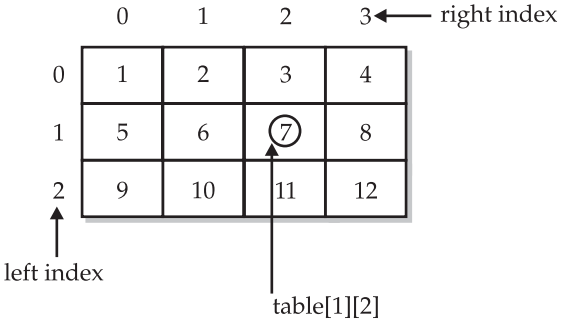


FIGURE 7-1 A conceptual view of the **table** array created by the **TwoD** program

For example, the following declaration creates a 4×10×3 three-dimensional integer array:

```
int[,,] multidim = new int[4, 10, 3];
```

To assign element 2, 4, 1 of **multidim** the value 100, use this statement:

```
multidim[2, 4, 1] = 100;
```

Here is a program that uses a three-dimensional array that holds a 3×3×3 matrix of values. It then sums the value on one of the diagonals through the cube.

```
// Sum the values on a diagonal of a 3x3x3 matrix.

using System;

class ThreeDMatrix {
  static void Main() {
    int[,,] m = new int[3, 3, 3];
    int sum = 0;
    int n = 1;

    for(int x=0; x < 3; x++)
      for(int y=0; y < 3; y++)
        for(int z=0; z < 3; z++)
          m[x, y, z] = n++;

    sum = m[0, 0, 0] + m[1, 1, 1] + m[2, 2, 2];

    Console.WriteLine("Sum of first diagonal: " + sum);
  }
}
```

The output is shown here:

```
Sum of first diagonal: 42
```

Initializing Multidimensional Arrays

A multidimensional array can be initialized by enclosing each dimension's initializer list within its own set of curly braces. For example, the general form of array initialization for a two-dimensional array is shown here:

type[,] *array_name* = {
  { *val*, *val*, *val*, ..., *val* },
  { *val*, *val*, *val*, ..., *val* },
.
.
.
  { *val*, *val*, *val*, ..., *val* }
};

Here, *val* indicates an initialization value. Each inner block designates a row. Within each row, the first value will be stored in the first position, the second value in the second position, and so on. Notice that commas separate the initializer blocks and that a semicolon follows the closing }.

For example, the following program initializes an array called **sqrs** with the numbers 1 through 10 and their squares.

```
// Initialize a two-dimensional array.

using System;

class Squares {
  static void Main() {
    int[,] sqrs = {
      { 1, 1 },
      { 2, 4 },
      { 3, 9 },
      { 4, 16 },
      { 5, 25 },
      { 6, 36 },
      { 7, 49 },
      { 8, 64 },
      { 9, 81 },
      { 10, 100 }
    };
    int i, j;

    for(i=0; i < 10; i++) {
      for(j=0; j < 2; j++)
        Console.Write(sqrs[i,j] + " ");
      Console.WriteLine();
    }
  }
}
```

Here is the output from the program:

```
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100
```

Jagged Arrays

In the preceding examples, when you created a two-dimensional array, you were creating what C# calls a *rectangular array.* Thinking of two-dimensional arrays as tables, a rectangular array is a two-dimensional array in which the length of each row is the same for the entire array. However, C# also allows you to create a special type of two-dimensional array called a *jagged array.* A jagged array is an *array of arrays* in which the length of each array can differ. Thus, a jagged array can be used to create a table in which the lengths of the rows are not the same.

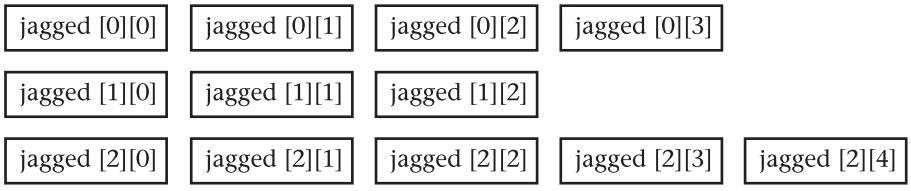
Jagged arrays are declared by using sets of square brackets to indicate each dimension. For example, to declare a two-dimensional jagged array, you will use this general form:

type[ ] [ ] *array-name* = new *type*[*size*][ ];

Here, *size* indicates the number of rows in the array. The rows, themselves, have not been allocated. Instead, the rows are allocated individually. This allows for the length of each row to vary. For example, the following code allocates memory for the first dimension of **jagged** when it is declared. It then allocates the second dimensions manually.

```
int[][] jagged = new int[3][];
jagged[0] = new int[4];
jagged[1] = new int[3];
jagged[2] = new int[5];
```

After this sequence executes, **jagged** looks like this:



It is easy to see how jagged arrays got their name!

Once a jagged array has been created, an element is accessed by specifying each index within its own set of brackets. For example, to assign the value 10 to element 2, 1 of **jagged**, you would use this statement:

```
jagged[2][1] = 10;
```

Note that this differs from the syntax that is used to access an element of a rectangular array.

The following program demonstrates the creation of a jagged two-dimensional array:

```
// Demonstrate jagged arrays.

using System;

class Jagged {
  static void Main() {
    int[][] jagged = new int[3][];
    jagged[0] = new int[4];
    jagged[1] = new int[3];
    jagged[2] = new int[5];

    int i;

    // Store values in first array.
    for(i=0; i < 4; i++)
      jagged[0][i] = i;

    // Store values in second array.
    for(i=0; i < 3; i++)
      jagged[1][i] = i;
```

```
    // Store values in third array.
    for(i=0; i < 5; i++)
      jagged[2][i] = i;

    // Display values in first array.
    for(i=0; i < 4; i++)
      Console.Write(jagged[0][i] + " ");

    Console.WriteLine();

    // Display values in second array.
    for(i=0; i < 3; i++)
      Console.Write(jagged[1][i] + " ");

    Console.WriteLine();

    // Display values in third array.
    for(i=0; i < 5; i++)
      Console.Write(jagged[2][i] + " ");

    Console.WriteLine();
  }
}
```

The output is shown here:

```
0 1 2 3
0 1 2
0 1 2 3 4
```

Jagged arrays are not used by all applications, but they can be effective in some situations. For example, if you need a very large two-dimensional array that is sparsely populated (that is, one in which not all of the elements will be used), then a jagged array might be a perfect solution.

One last point: Because jagged arrays are arrays of arrays, there is no restriction that requires that the arrays be one-dimensional. For example, the following creates an array of two-dimensional arrays:

```
int[][,] jagged = new int[3][,];
```

The next statement assigns **jagged[0]** a reference to a 4×2 array:

```
jagged[0] = new int[4, 2];
```

The following statement assigns a value to **jagged[0][1,0]**:

```
jagged[0][1,0] = i;
```

Assigning Array References

As with other objects, when you assign one array reference variable to another, you are simply making both variables refer to the same array. You are neither causing a copy of the

array to be created, nor are you causing the contents of one array to be copied to the other. For example, consider this program:

```
// Assigning array reference variables.

using System;

class AssignARef {
  static void Main() {
    int i;

    int[] nums1 = new int[10];
    int[] nums2 = new int[10];

    for(i=0; i < 10; i++) nums1[i] = i;

    for(i=0; i < 10; i++) nums2[i] = -i;

    Console.Write("Here is nums1: ");
    for(i=0; i < 10; i++)
      Console.Write(nums1[i] + " ");
    Console.WriteLine();

    Console.Write("Here is nums2: ");
    for(i=0; i < 10; i++)
      Console.Write(nums2[i] + " ");
    Console.WriteLine();

    nums2 = nums1; // now nums2 refers to nums1

    Console.Write("Here is nums2 after assignment: ");
    for(i=0; i < 10; i++)
      Console.Write(nums2[i] + " ");
    Console.WriteLine();

    // Next, operate on nums1 array through nums2.
    nums2[3] = 99;

    Console.Write("Here is nums1 after change through nums2: ");
    for(i=0; i < 10; i++)
      Console.Write(nums1[i] + " ");
    Console.WriteLine();
  }
}
```

The output from the program is shown here:

```
Here is nums1: 0 1 2 3 4 5 6 7 8 9
Here is nums2: 0 -1 -2 -3 -4 -5 -6 -7 -8 -9
Here is nums2 after assignment: 0 1 2 3 4 5 6 7 8 9
Here is nums1 after change through nums2: 0 1 2 99 4 5 6 7 8 9
```

As the output shows, after the assignment of **nums1** to **nums2**, both array reference variables refer to the same object.

Using the Length Property

A number of benefits result because C# implements arrays as objects. One comes from the fact that each array has associated with it a **Length** property that contains the number of elements that an array can hold. Thus, each array provides a means by which its length can be determined. Here is a program that demonstrates this property:

```
// Use the Length array property.

using System;

class LengthDemo {
  static void Main() {
    int[] nums = new int[10];

    Console.WriteLine("Length of nums is " + nums.Length);

    // Use Length to initialize nums.
    for(int i=0; i < nums.Length; i++)
      nums[i] = i * i;

    // Now use Length to display nums.
    Console.Write("Here is nums: ");
    for(int i=0; i < nums.Length; i++)
      Console.Write(nums[i] + " ");

    Console.WriteLine();
  }
}
```

This program displays the following output:

```
Length of nums is 10
Here is nums: 0 1 4 9 16 25 36 49 64 81
```

In **LengthDemo** notice the way that **nums.Length** is used by the **for** loops to govern the number of iterations that take place. Since each array carries with it its own length, you can use this information rather than manually keeping track of an array's size. Keep in mind that the value of **Length** has nothing to do with the number of elements that are actually in use. **Length** contains the number of elements that the array is capable of holding.

When the length of a multidimensional array is obtained, the total number of elements that can be held by the array is returned. For example:

```
// Use the Length array property on a 3D array.

using System;

class LengthDemo3D {
  static void Main() {
    int[,,] nums = new int[10, 5, 6];

    Console.WriteLine("Length of nums is " + nums.Length);
  }
}
```

The output is shown here:

```
Length of nums is 300
```

As the output verifies, **Length** obtains the number of elements that **nums** can hold, which is 300 (10×5×6) in this case. It is not possible to use **Length** to obtain the length of a specific dimension.

The inclusion of the **Length** property simplifies many algorithms by making certain types of array operations easier—and safer—to perform. For example, the following program uses **Length** to reverse the contents of an array by copying it back-to-front into another array:

```
// Reverse an array.

using System;

class RevCopy {
  static void Main() {
    int i,j;
    int[] nums1 = new int[10];
    int[] nums2 = new int[10];

    for(i=0; i < nums1.Length; i++) nums1[i] = i;

    Console.Write("Original contents: ");
    for(i=0; i < nums2.Length; i++)
      Console.Write(nums1[i] + " ");

    Console.WriteLine();

    // Reverse copy nums1 to nums2.
    if(nums2.Length >= nums1.Length) // make sure nums2 is long enough
      for(i=0, j=nums1.Length-1; i < nums1.Length; i++, j--)
        nums2[j] = nums1[i];

    Console.Write("Reversed contents: ");
    for(i=0; i < nums2.Length; i++)
      Console.Write(nums2[i] + " ");

    Console.WriteLine();
  }
}
```

Here is the output:

```
Original contents: 0 1 2 3 4 5 6 7 8 9
Reversed contents: 9 8 7 6 5 4 3 2 1 0
```

Here, **Length** helps perform two important functions. First, it is used to confirm that the target array is large enough to hold the contents of the source array. Second, it provides the termination condition of the **for** loop that performs the reverse copy. Of course, in this simple example, the size of the arrays is easily known, but this same approach can be applied to a wide range of more challenging situations.

Using Length with Jagged Arrays

A special case occurs when **Length** is used with jagged arrays. In this situation, it is possible to obtain the length of each individual array. For example, consider the following program, which simulates the CPU activity on a network with four nodes:

```
// Demonstrate Length with jagged arrays.

using System;

class Jagged {
  static void Main() {
    int[][] network_nodes = new int[4][];
    network_nodes[0] = new int[3];
    network_nodes[1] = new int[7];
    network_nodes[2] = new int[2];
    network_nodes[3] = new int[5];


    int i, j;

    // Fabricate some fake CPU usage data.
    for(i=0; i < network_nodes.Length; i++)
      for(j=0; j < network_nodes[i].Length; j++)
        network_nodes[i][j] = i * j + 70;


    Console.WriteLine("Total number of network nodes: " +
                      network_nodes.Length + "\n");

    for(i=0; i < network_nodes.Length; i++) {
      for(j=0; j < network_nodes[i].Length; j++) {
        Console.Write("CPU usage at node " + i +
                      " CPU " + j + ": ");
        Console.Write(network_nodes[i][j] + "% ");
        Console.WriteLine();
      }
      Console.WriteLine();
    }
  }
}
```

The output is shown here:

```
Total number of network nodes: 4

CPU usage at node 0 CPU 0: 70%
CPU usage at node 0 CPU 1: 70%
CPU usage at node 0 CPU 2: 70%

CPU usage at node 1 CPU 0: 70%
CPU usage at node 1 CPU 1: 71%
CPU usage at node 1 CPU 2: 72%
CPU usage at node 1 CPU 3: 73%
CPU usage at node 1 CPU 4: 74%
```

```
CPU usage at node 1 CPU 5: 75%
CPU usage at node 1 CPU 6: 76%

CPU usage at node 2 CPU 0: 70%
CPU usage at node 2 CPU 1: 72%

CPU usage at node 3 CPU 0: 70%
CPU usage at node 3 CPU 1: 73%
CPU usage at node 3 CPU 2: 76%
CPU usage at node 3 CPU 3: 79%
CPU usage at node 3 CPU 4: 82%
```

Pay special attention to the way **Length** is used on the jagged array **network_nodes**. Recall, a two-dimensional jagged array is an array of arrays. Thus, when the expression

```
network_nodes.Length
```

is used, it obtains the number of *arrays* stored in **network_nodes**, which is four in this case. To obtain the length of any individual array in the jagged array, you will use an expression such as this:

```
network_nodes[0].Length
```

which, in this case, obtains the length of the first array.

Implicitly Typed Arrays

As explained in Chapter 3, C# 3.0 adds the ability to declare implicitly typed variables by using the **var** keyword. These are variables whose type is determined by the compiler, based on the type of the initializing expression. Thus, all implicitly typed variables must be initialized. Using the same mechanism, it is also possible to create an implicitly typed array. As a general rule, implicitly typed arrays are for use in certain types of queries involving LINQ, which is described in Chapter 19. In most other cases, you will use the "normal" array declaration approach. Implicitly typed arrays are introduced here for completeness.

An implicitly typed array is declared using the keyword **var**, but you *do not* follow **var** with **[ ]**. Furthermore, the array must be initialized because it is the type of the initializers that determine the element type of the array. All of the initializers must be of the same or compatible type. Here is an example of an implicitly typed array:

```
var vals = new[] { 1, 2, 3, 4, 5 };
```

This creates an array of **int** that is five elements long. A reference to that array is assigned to **vals**. Thus, the type of **vals** is "array of **int**" and it has five elements. Again, notice that **var** is not followed by **[ ]**. Also, even though the array is being initialized, you must include **new[ ]**. It's not optional in this context.

Here is another example. It creates a two-dimensional array of **double**:

```
var vals = new[,] { {1.1, 2.2}, {3.3, 4.4},{ 5.5, 6.6} };
```

In this case, **vals** has the dimensions 2×3.

You can also declare implicitly typed jagged arrays. For example, consider the following program:

```
// Demonstrate an implicitly typed jagged array.

using System;

class Jagged {
  static void Main() {

    var jagged = new[] {
       new[] { 1, 2, 3, 4 },
       new[] { 9, 8, 7 },
       new[] { 11, 12, 13, 14, 15 }
    };

    for(int j = 0; j < jagged.Length; j++) {
      for(int i=0; i < jagged[j].Length; i++)
        Console.Write(jagged[j][i] + " ");

      Console.WriteLine();
    }
  }
}
```

The program produces the following output:

```
1 2 3 4
9 8 7
11 12 13 14 15
```

Pay special attention to the declaration of **jagged**:

```
var jagged = new[] {
   new[] { 1, 2, 3, 4 },
   new[] { 9, 8, 7 },
   new[] { 11, 12, 13, 14, 15 }
};
```

Notice how **new[ ]** is used in two ways. First, it creates the array of arrays. Second, it creates each individual array, based on the number and type of initializers. As you would expect, all of the initializers in the individual arrays must be of the same type. The same general approach used to declare **jagged** can be used to declare any implicitly typed jagged array.

As mentioned, implicitly typed arrays are most applicable to LINQ-based queries. They are not meant for general use. In most cases, you should use explicitly typed arrays.

The foreach Loop

In Chapter 5, it was mentioned that C# defines a loop called **foreach**, but a discussion of that statement was deferred until later. The time for that discussion has now come.

The **foreach** loop is used to cycle through the elements of a *collection.* A collection is a group of objects. C# defines several types of collections, of which one is an array. The general form of **foreach** is shown here:

foreach(*type loopvar* in *collection*) *statement*;

Here, *type loopvar* specifies the type and name of an *iteration variable.* The iteration variable receives the value of the next element in the collection each time the **foreach** loop iterates. The collection being cycled through is specified by *collection*, which, for the rest of this discussion, is an array. Thus, *type* must be the same as (or compatible with) the element type of the array. Beginning with C# 3.0, *type* can also be **var**, in which case the compiler determines the type based on the element type of the array. This can be useful when working with certain queries, as described later in this book. Normally, you will explicitly specify the type.

Here is how **foreach** works. When the loop begins, the first element in the array is obtained and assigned to *loopvar.* Each subsequent iteration obtains the next element from the array and stores it in *loopvar.* The loop ends when there are no more elements to obtain. Thus, the **foreach** cycles through the array one element at a time, from start to finish.

One important point to remember about **foreach** is that the iteration variable *loopvar* is read-only. This means you can't change the contents of an array by assigning the iteration variable a new value.

Here is a simple example that uses **foreach**. It creates an array of integers and gives it some initial values. It then displays those values, computing the summation in the process.

```
// Use the foreach loop.

using System;

class ForeachDemo {
  static void Main() {
    int sum = 0;
    int[] nums = new int[10];

    // Give nums some values.
    for(int i = 0; i < 10; i++)
      nums[i] = i;

    // Use foreach to display and sum the values.
    foreach(int x in nums) {
      Console.WriteLine("Value is: " + x);
      sum += x;
    }
    Console.WriteLine("Summation: " + sum);
  }
}
```

The output from the program is shown here:

```
Value is: 0
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 6
Value is: 7
```

```
Value is: 8
Value is: 9
Summation: 45
```

As this output shows, **foreach** cycles through an array in sequence from the lowest index to the highest.

Although the **foreach** loop iterates until all elements in an array have been examined, it is possible to terminate a **foreach** loop early by using a **break** statement. For example, this program sums only the first five elements of **nums**:

```
// Use break with a foreach.

using System;

class ForeachDemo {
  static void Main() {
    int sum = 0;
    int[] nums = new int[10];

    // Give nums some values.
    for(int i = 0; i < 10; i++)
      nums[i] = i;

    // Use foreach to display and sum the values.
    foreach(int x in nums) {
      Console.WriteLine("Value is: " + x);
      sum += x;
      if(x == 4) break; // stop the loop when 4 is obtained
    }
    Console.WriteLine("Summation of first 5 elements: " + sum);
  }
}
```

This is the output produced:

```
Value is: 0
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Summation of first 5 elements: 10
```

As is evident, the **foreach** loop stops after the fifth element has been obtained.

The **foreach** loop also works on multidimensional arrays. It returns those elements in row order, from first to last.

```
// Use foreach on a two-dimensional array.

using System;

class ForeachDemo2 {
  static void Main() {
    int sum = 0;
    int[,] nums = new int[3,5];
```

```
    // Give nums some values.
    for(int i = 0; i < 3; i++)
      for(int j=0; j < 5; j++)
        nums[i,j] = (i+1)*(j+1);

    // Use foreach to display and sum the values.
    foreach(int x in nums) {
      Console.WriteLine("Value is: " + x);
      sum += x;
    }
    Console.WriteLine("Summation: " + sum);
  }
}
```

The output from this program is shown here:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 2
Value is: 4
Value is: 6
Value is: 8
Value is: 10
Value is: 3
Value is: 6
Value is: 9
Value is: 12
Value is: 15
Summation: 90
```

Since the **foreach** loop can only cycle through an array sequentially, from start to finish, you might think that its use is limited. However, this is not true. A large number of algorithms require exactly this mechanism, of which one of the most common is searching. For example, the following program uses a **foreach** loop to search an array for a value. It stops if the value is found.

```
// Search an array using foreach.

using System;

class Search {
  static void Main() {
    int[] nums = new int[10];
    int val;
    bool found = false;

    // Give nums some values.
    for(int i = 0; i < 10; i++)
      nums[i] = i;

    val = 5;
```

```
    // Use foreach to search nums for key.
    foreach(int x in nums) {
      if(x == val) {
        found = true;
        break;
      }
    }

    if(found)
      Console.WriteLine("Value found!");
  }
}
```

The output is shown here:

```
Value found!
```

The **foreach** loop is an excellent choice in this application because searching an array involves examining each element. Other types of **foreach** applications include such things as computing an average, finding the minimum or maximum of a set, looking for duplicates, and so on. As you will see later in this book, **foreach** is especially useful when operating on other types of collections.

Strings

From a day-to-day programming standpoint, one of the most important of C#'s data types is **string**. **string** defines and supports character strings. In many other programming languages, a string is an array of characters. This is not the case with C#. In C#, strings are objects. Thus, **string** is a reference type. Although **string** is a built-in data type in C#, a discussion of **string** needed to wait until classes and objects had been introduced.

Actually, you have been using the **string** class since Chapter 2, but you did not know it. When you create a string literal, you are actually creating a **string** object. For example, in the statement

```
Console.WriteLine("In C#, strings are objects.");
```

the string "In C#, strings are objects." is automatically made into a **string** object by C#. Thus, the use of the **string** class has been "below the surface" in the preceding programs. In this section, you will learn to handle them explicitly.

Constructing Strings

The easiest way to construct a **string** is to use a string literal. For example, here **str** is a **string** reference variable that is assigned a reference to a string literal:

```
string str = "C# strings are powerful.";
```

In this case, **str** is initialized to the character sequence "C# strings are powerful."

You can also create a **string** from a **char** array. For example:

```
char[] charray = {'t', 'e', 's', 't'};
string str = new string(charray);
```

Once you have created a **string** object, you can use it nearly anywhere that a quoted string is allowed. For example, you can use a **string** object as an argument to **WriteLine( )**, as shown in this example:

```
// Introduce string.

using System;

class StringDemo {
  static void Main() {

    char[] charray = {'A', ' ', 's', 't', 'r', 'i', 'n', 'g', '.' };
    string str1 = new string(charray);
    string str2 = "Another string.";

    Console.WriteLine(str1);
    Console.WriteLine(str2);
  }
}
```

The output from the program is shown here:

```
A string.
Another string.
```

Operating on Strings

The **string** class contains several methods that operate on strings. Table 7-1 shows a few. The **string** type also includes the **Length** property, which contains the length of the string.

To obtain the value of an individual character of a string, you simply use an index. For example:

```
string str = "test";
Console.WriteLine(str[0]);
```

This displays "t", the first character of "test". Like arrays, string indexes begin at zero. One important point, however, is that you cannot assign a new value to a character within a string using an index. An index can only be used to obtain a character.

Method	Description
static string Copy(string *str*)	Returns a copy of *str*.
int CompareTo(string *str*)	Returns less than zero if the invoking string is less than *str*, greater than zero if the invoking string is greater than *str*, and zero if the strings are equal.
int IndexOf(string *str*)	Searches the invoking string for the substring specified by *str*. Returns the index of the first match, or –1 on failure.
int LastIndexOf(string *str*)	Searches the invoking string for the substring specified by *str*. Returns the index of the last match, or –1 on failure.
string ToLower( )	Returns a lowercase version of the invoking string.
string ToUpper( )	Returns an uppercase version of the invoking string.

Table 7-1 Some Common String Handling Methods

To test two strings for equality, you can use the **==** operator. Normally, when the **==** operator is applied to object references, it determines if both references refer to the same object. This differs for objects of type **string**. When the **==** is applied to two **string** references, the contents of the strings, themselves, are compared for equality. The same is true for the **!=** operator: When comparing **string** objects, the contents of the strings are compared. For other types of string comparisons, you will need to use the **CompareTo()** method.

Here is a program that demonstrates several string operations:

```
// Some string operations.

using System;

class StrOps {
  static void Main() {
    string str1 =
      "When it comes to .NET programming, C# is #1.";
    string str2 = string.Copy(str1);
    string str3 = "C# strings are powerful.";
    string strUp, strLow;
    int result, idx;

    Console.WriteLine("str1: " + str1);

    Console.WriteLine("Length of str1: " +
                      str1.Length);

    // Create upper- and lowercase versions of str1.
    strLow = str1.ToLower();
    strUp =  str1.ToUpper();
    Console.WriteLine("Lowercase version of str1:\n    " +
                      strLow);
    Console.WriteLine("Uppercase version of str1:\n    " +
                      strUp);

    Console.WriteLine();

    // Display str1, one char at a time.
    Console.WriteLine("Display str1, one char at a time.");
    for(int i=0; i < str1.Length; i++)
      Console.Write(str1[i]);
    Console.WriteLine("\n");

    // Compare strings.
    if(str1 == str2)
      Console.WriteLine("str1 == str2");
    else
      Console.WriteLine("str1 != str2");

    if(str1 == str3)
      Console.WriteLine("str1 == str3");
    else
      Console.WriteLine("str1 != str3");
```

```
    result = str1.CompareTo(str3);
    if(result == 0)
      Console.WriteLine("str1 and str3 are equal");
    else if(result < 0)
      Console.WriteLine("str1 is less than str3");
    else
      Console.WriteLine("str1 is greater than str3");

    Console.WriteLine();

    // Assign a new string to str2.
    str2 = "One Two Three One";

    // Search a string.
    idx = str2.IndexOf("One");
    Console.WriteLine("Index of first occurrence of One: " + idx);
    idx = str2.LastIndexOf("One");
    Console.WriteLine("Index of last occurrence of One: " + idx);

  }
}
```

This program generates the following output:

```
str1: When it comes to .NET programming, C# is #1.
Length of str1: 44
Lowercase version of str1:
    when it comes to .net programming, c# is #1.
Uppercase version of str1:
    WHEN IT COMES TO .NET PROGRAMMING, C# IS #1.

Display str1, one char at a time.
When it comes to .NET programming, C# is #1.

str1 == str2
str1 != str3
str1 is greater than str3

Index of first occurrence of One: 0
Index of last occurrence of One: 14
```

You can concatenate (join together) two strings using the **+** operator. For example, this statement:

```
string str1 = "One";
string str2 = "Two";
string str3 = "Three";
string str4 = str1 + str2 + str3;
```

initializes **str4** with the string "OneTwoThree".

One other point: The **string** keyword is an *alias* for (that is, maps directly to) the **System.String** class defined by the .NET Framework class library. Thus, the fields and methods defined by **string** are those of the **System.String** class, which includes more than the sampling described here. **System.String** is examined in detail in Part II.

Arrays of Strings

Like any other data type, strings can be assembled into arrays. For example:

```
// Demonstrate string arrays.
using System;

class StringArrays {
  static void Main() {
    string[] str = { "This", "is", "a", "test." };

    Console.WriteLine("Original array: ");
    for(int i=0; i < str.Length; i++)
      Console.Write(str[i] + " ");
    Console.WriteLine("\n");

    // Change a string.
    str[1] = "was";
    str[3] = "test, too!";

    Console.WriteLine("Modified array: ");
    for(int i=0; i < str.Length; i++)
      Console.Write(str[i] + " ");
  }
}
```

Here is the output from this program:

```
Original array:
This is a test.

Modified array:
This was a test, too!
```

Here is a more interesting example. The following program displays an integer value using words. For example, the value 19 will display as "one nine".

```
// Display the digits of an integer using words.

using System;

class ConvertDigitsToWords {
  static void Main() {
    int num;
    int nextdigit;
    int numdigits;
    int[] n = new int[20];

    string[] digits = { "zero", "one", "two",
                        "three", "four", "five",
                        "six", "seven", "eight",
                        "nine" };

    num = 1908;
```

```
    Console.WriteLine("Number: " + num);

    Console.Write("Number in words: ");

    nextdigit = 0;
    numdigits = 0;

    // Get individual digits and store in n.
    // These digits are stored in reverse order.
    do {
      nextdigit = num % 10;
      n[numdigits] = nextdigit;
      numdigits++;
      num = num / 10;
    } while(num > 0);
    numdigits--;

    // Display the words.
    for( ; numdigits >= 0; numdigits--)
      Console.Write(digits[n[numdigits]] + " ");

    Console.WriteLine();
  }
}
```

The output is shown here:

```
Number: 1908
Number in words: one nine zero eight
```

In the program, the **string** array **digits** holds in order the word equivalents of the digits from zero to nine. The program converts an integer into words by first obtaining each digit of the value and then storing those digits, in reverse order, in the **int** array called **n**. Then, this array is cycled through from back to front. In the process, each integer value in **n** is used as an index into **digits**, with the corresponding string being displayed.

Strings Are Immutable

Here is something that might surprise you: The contents of a **string** object are immutable. That is, once created, the character sequence comprising that string cannot be altered. This restriction allows strings to be implemented more efficiently. Even though this probably sounds like a serious drawback, it isn't. When you need a string that is a variation on one that already exists, simply create a new string that contains the desired changes. Since unused string objects are automatically garbage-collected, you don't even need to worry about what happens to the discarded strings.

It must be made clear, however, that **string** reference variables may, of course, change which object they refer to. It is just that the contents of a specific **string** object cannot be changed after it is created.

To fully understand why immutable strings are not a hindrance, we will use another of **string**'s methods: **Substring( )**. The **Substring( )** method returns a new string that contains a specified portion of the invoking string. Because a new **string** object is manufactured that

contains the substring, the original string is unaltered, and the rule of immutability is still intact. The form of **Substring( )** that we will be using is shown here:

string Substring(int *start*, int *len*)

Here, *start* specifies the beginning index, and *len* specifies the length of the substring.

Here is a program that demonstrates **Substring( )** and the principle of immutable strings:

```
// Use Substring().

using System;

class SubStr {
  static void Main() {
    string orgstr = "C# makes strings easy.";

    // construct a substring
    string substr = orgstr.Substring(5, 12);

    Console.WriteLine("orgstr: " + orgstr);
    Console.WriteLine("substr: " + substr);
  }
}
```

Here is the output from the program:

```
orgstr: C# makes strings easy.
substr: kes strings
```

As you can see, the original string **orgstr** is unchanged and **substr** contains the substring.

One more point: Although the immutability of **string** objects is not usually a restriction or hindrance, there may be times when it would be beneficial to modify a string. To allow this, C# offers a class called **StringBuilder**, which is in the **System.Text** namespace. It creates string objects that can be changed. For most purposes, however, you will want to use **string**, not **StringBuilder**.

Strings Can Be Used in switch Statements

A **string** can be used to control a **switch** statement. It is the only non-integer type that can be used in the **switch**. The fact that strings can be used in **switch** statements makes it possible to handle some otherwise challenging situations more easily than you might expect. For example, the following program displays the digit equivalent of the words "one," "two," and "three":

```
// A string can control a switch statement.

using System;

class StringSwitch {
  static void Main() {
    string[] strs = { "one", "two", "three", "two", "one" };
```

```
    foreach(string s in strs) {
      switch(s) {
        case "one":
          Console.Write(1);
          break;
        case "two":
          Console.Write(2);
          break;
        case "three":
          Console.Write(3);
          break;
      }
    }
    Console.WriteLine();
  }
}
```

The output is shown here:

```
12321
```

This page intentionally left blank

CHAPTER 8

A Closer Look at Methods and Classes

This chapter resumes the examination of classes and methods. It begins by explaining how to control access to the members of a class. It then discusses the passing and returning of objects, method overloading, the various forms of **Main()**, recursion, and the use of the keyword **static**.

Controlling Access to Class Members

In its support for encapsulation, the class provides two major benefits. First, it links data with code. You have been taking advantage of this aspect of the class since Chapter 6. Second, it provides the means by which access to members can be controlled. It is this second feature that is examined here.

Although C#'s approach is a bit more sophisticated, in essence, there are two basic types of class members: public and private. A public member can be freely accessed by code defined outside of its class. This is the type of class member that we have been using up to this point. A private member can be accessed only by methods defined by its class. It is through the use of private members that access is controlled.

Restricting access to a class' members is a fundamental part of object-oriented programming because it helps prevent the misuse of an object. By allowing access to private data only through a well-defined set of methods, you can prevent improper values from being assigned to that data—by performing a range check, for example. It is not possible for code outside the class to set the value of a private member directly. You can also control precisely how and when the data within an object is used. Thus, when correctly implemented, a class creates a "black box" that can be used, but the inner workings of which are not open to tampering.

C#'s Access Modifiers

Member access control is achieved through the use of four *access modifiers*: **public**, **private**, **protected**, and **internal**. In this chapter, we will be concerned with **public** and **private**. The **protected** modifier applies only when inheritance is involved and is described in Chapter 11.

The **internal** modifier applies mostly to the use of an *assembly*, which for C# loosely means a deployable program or library. The **internal** modifier is examined in Chapter 16.

When a member of a class is modified by the **public** specifier, that member can be accessed by any other code in your program. This includes methods defined inside other classes.

When a member of a class is specified as **private**, then that member can be accessed only by other members of its class. Thus, methods in other classes are not able to access a **private** member of another class. As explained in Chapter 6, if no access specifier is used, a class member is private to its class by default. Thus, the **private** specifier is optional when creating private class members.

An access specifier precedes the rest of a member's type specification. That is, it must begin a member's declaration statement. Here are some examples:

```
public string errMsg;
private double bal;
private bool isError(byte status) { // ...
```

To understand the difference between **public** and **private**, consider the following program:

```
// Public vs. private access.

using System;

class MyClass {
  private int alpha; // private access explicitly specified
  int beta;          // private access by default
  public int gamma;  // public access

  // Methods to access alpha and beta. It is OK for a member
  // of a class to access a private member of the same class.

  public void SetAlpha(int a) {
    alpha = a;
  }

  public int GetAlpha() {
    return alpha;
  }

  public void SetBeta(int a) {
    beta = a;
  }

  public int GetBeta() {
    return beta;
  }
}

class AccessDemo {
  static void Main() {
    MyClass ob = new MyClass();
```

```
    // Access to alpha and beta is allowed only through methods.
    ob.SetAlpha(-99);
    ob.SetBeta(19);
    Console.WriteLine("ob.alpha is " + ob.GetAlpha());
    Console.WriteLine("ob.beta is " + ob.GetBeta());

    // You cannot access alpha or beta like this:
//  ob.alpha = 10; // Wrong! alpha is private!
//  ob.beta = 9;   // Wrong! beta is private!

    // It is OK to directly access gamma because it is public.
    ob.gamma = 99;
   }
}
```

As you can see, inside the **MyClass** class, **alpha** is specified as **private**, **beta** is private by default, and **gamma** is specified as **public**. Because **alpha** and **beta** are private, they cannot be accessed by code outside of their class. Therefore, inside the **AccessDemo** class, neither can be used directly. Each must be accessed through public methods, such as **SetAlpha()** and **GetAlpha()**. For example, if you were to remove the comment symbol from the beginning of the following line

```
//  ob.alpha = 10; // Wrong! alpha is private!
```

you would not be able to compile this program because of the access violation. Although access to **alpha** by code outside of **MyClass** is not allowed, methods defined within **MyClass** can freely access it, as the **SetAlpha()** and **GetAlpha()** methods show. The same is true for **beta**.

The key point is this: A private member can be used freely by other members of its class, but it cannot be accessed by code outside its class.

Applying Public and Private Access

The proper use of public and private access is a key component of successful object-oriented programming. Although there are no hard and fast rules, here are some general principles that serve as guidelines:

- Members of a class that are used only within the class itself should be private.
- Instance data that must be within a specific range should be private, with access provided through public methods that can perform range checks.
- If changing a member can cause an effect that extends beyond the member itself (that is, affects other aspects of the object), that member should be private, and access to it should be controlled.
- Members that can cause harm to an object when improperly used should be private. Access to these members should be through public methods that prevent improper usage.
- Methods that get and set the values of private data must be public.
- Public instance variables are permissible when there is no reason for them to be private.

Of course, there are many nuances that the preceding rules do not address, and special cases cause one or more rules to be violated. But, in general, if you follow these rules, you will be creating resilient objects that are not easily misused.

Controlling Access: A Case Study

To better understand the "how and why" behind access control, a case study is useful. One of the quintessential examples of object-oriented programming is a class that implements a stack. As you probably know, a *stack* is a data structure that implements a last-in, first-out list. Its name comes from the analogy of a stack of plates on a table. The first plate on the table is the last one to be used.

A stack is a classic example of object-oriented programming because it combines storage for information along with the methods that access that information. Thus, a stack is a *data engine* that enforces the last-in, first-out usage. Such a combination is an excellent choice for a class in which the members that provide storage for the stack are private, and public methods provide access. By encapsulating the underlying storage, it is not possible for code that uses the stack to access the elements out of order.

A stack defines two basic operations: *push* and *pop.* A push puts a value onto the top of the stack. A pop removes a value from the top of the stack. Thus, a pop is consumptive; once a value has been popped off the stack, it has been removed and cannot be accessed again.

The example shown here creates a class called **Stack** that implements a stack. The underlying storage for the stack is provided by a private array. The push and pop operations are available through the public methods of the **Stack** class. Thus, the public methods enforce the last-in, first-out mechanism. As shown here, the **Stack** class stores characters, but the same mechanism could be used to store any type of data:

```
// A stack class for characters.

using System;

class Stack {
  // These members are private.
  char[] stck; // holds the stack
  int tos;     // index of the top of the stack

  // Construct an empty Stack given its size.
  public Stack(int size) {
    stck = new char[size]; // allocate memory for stack
    tos = 0;
  }

  // Push characters onto the stack.
  public void Push(char ch) {
    if(tos==stck.Length) {
      Console.WriteLine(" -- Stack is full.");
      return;
    }

    stck[tos] = ch;
    tos++;
  }

  // Pop a character from the stack.
```

```
  public char Pop() {
    if(tos==0) {
      Console.WriteLine(" -- Stack is empty.");
      return (char) 0;
    }

    tos--;
    return stck[tos];
  }

  // Return true if the stack is full.
  public bool IsFull() {
    return tos==stck.Length;
  }

  // Return true if the stack is empty.
  public bool IsEmpty() {
    return tos==0;
  }

  // Return total capacity of the stack.
  public int Capacity() {
    return stck.Length;
  }

  // Return number of objects currently on the stack.
  public int GetNum() {
    return tos;
  }
}
```

Let's examine this class closely. The **Stack** class begins by declaring these two instance variables:

```
// These members are private.
char[] stck; // holds the stack
int tos;     // index of the top of the stack
```

The **stck** array provides the underlying storage for the stack, which in this case holds characters. Notice that no array is allocated. The allocation of the actual array is handled by the **Stack** constructor. The **tos** member holds the index of the top of the stack.

Both the **tos** and **stck** members are private. This enforces the last-in, first-out stack mechanism. If public access to **stck** were allowed, then the elements on the stack could be accessed out of order. Also, since **tos** holds the index of the top element in the stack, manipulations of **tos** by code outside the **Stack** class must be prevented in order to avoid corruption of the stack. Access to **stck** and **tos** is available, indirectly, to the user of **Stack** through the various public methods described shortly.

The stack constructor is shown next:

```
// Construct an empty Stack given its size.
public Stack(int size) {
  stck = new char[size]; // allocate memory for stack
  tos = 0;
}
```

The constructor is passed the desired size of the stack. It allocates the underlying array and sets **tos** to zero. Thus, a zero value in **tos** indicates that the stack is empty.

The public **Push()** method puts an element onto the stack. It is shown here:

```
// Push characters onto the stack.
public void Push(char ch) {
  if(tos==stck.Length) {
    Console.WriteLine(" -- Stack is full.");
    return;
  }

  stck[tos] = ch;
  tos++;
}
```

The element to be pushed onto the stack is passed in **ch**. Before the element is added to the stack, a check is made to ensure that there is still room in the underlying array. This is done by making sure that **tos** does not exceed the length of **stck**. If there is still room, the element is stored in **stck** at the index specified by **tos**, and then **tos** is incremented. Thus, **tos** always contains the index of the next free element in **stck**.

To remove an element from the stack, call the public method **Pop()**. It is shown here:

```
// Pop a character from the stack.
public char Pop() {
  if(tos==0) {
    Console.WriteLine(" -- Stack is empty.");
    return (char) 0;
  }

  tos--;
  return stck[tos];
}
```

Here, the value of **tos** is checked. If it is zero, the stack is empty. Otherwise, **tos** is decremented, and the element at that index is returned.

Although **Push()** and **Pop()** are the only methods needed to implement a stack, some others are quite useful, and the **Stack** class defines four more. These are **IsFull()**, **IsEmpty()**, **Capacity()**, and **GetNum()**, and they provide information about the state of the stack. They are shown here:

```
// Return true if the stack is full.
public bool IsFull() {
  return tos==stck.Length;
}

// Return true if the stack is empty.
public bool IsEmpty() {
  return tos==0;
}

// Return total capacity of the stack.
```

```
public int Capacity() {
  return stck.Length;
}

// Return number of objects currently on the stack.
public int GetNum() {
  return tos;
}
```

The **IsFull()** method returns **true** when the stack is full and **false** otherwise. The **IsEmpty()** method returns **true** when the stack is empty and **false** otherwise. To obtain the total capacity of the stack (that is, the total number of elements it can hold), call **Capacity()**. To obtain the number of elements currently stored on the stack, call **GetNum()**. These methods are useful because the information they provide requires access to **tos**, which is private. They are also examples of how public methods can provide safe access to private members.

The following program demonstrates the stack:

```
// Demonstrate the Stack class.

using System;

class StackDemo {
  static void Main() {
    Stack stk1 = new Stack(10);
    Stack stk2 = new Stack(10);
    Stack stk3 = new Stack(10);
    char ch;
    int i;

    // Put some characters into stk1.
    Console.WriteLine("Push A through J onto stk1.");
    for(i=0; !stk1.IsFull(); i++)
      stk1.Push((char) ('A' + i));

    if(stk1.IsFull()) Console.WriteLine("stk1 is full.");

    // Display the contents of stk1.
    Console.Write("Contents of stk1: ");
    while( !stk1.IsEmpty() ) {
      ch = stk1.Pop();
      Console.Write(ch);
    }

    Console.WriteLine();

    if(stk1.IsEmpty()) Console.WriteLine("stk1 is empty.\n");

    // Put more characters into stk1.
    Console.WriteLine("Again push A through J onto stk1.");
    for(i=0; !stk1.IsFull(); i++)
      stk1.Push((char) ('A' + i));

    // Now, pop from stk1 and push the element in stk2.
```

```
    // This causes stk2 to hold the elements in reverse order.
    Console.WriteLine("Now, pop chars from stk1 and push " +
                      "them onto stk2.");
    while( !stk1.IsEmpty() ) {
      ch = stk1.Pop();
      stk2.Push(ch);
    }

    Console.Write("Contents of stk2: ");
    while( !stk2.IsEmpty() ) {
      ch = stk2.Pop();
      Console.Write(ch);
    }

    Console.WriteLine("\n");

    // Put 5 characters into stack.
    Console.WriteLine("Put 5 characters on stk3.");
    for(i=0; i < 5; i++)
      stk3.Push((char) ('A' + i));

    Console.WriteLine("Capacity of stk3: " + stk3.Capacity());
    Console.WriteLine("Number of objects in stk3: " +
                      stk3.GetNum());
  }
}
```

The output from the program is shown here:

```
Push A through J onto stk1.
stk1 is full.
Contents of stk1: JIHGFEDCBA
stk1 is empty.

Again push A through J onto stk1.
Now, pop chars from stk1 and push them onto stk2.
Contents of stk2: ABCDEFGHIJ

Put 5 characters on stk3.
Capacity of stk3: 10
Number of objects in stk3: 5
```

Pass References to Methods

Up to this point, the examples in this book have been using value types, such as **int** or **double**, as parameters to methods. However, it is both correct and common to use a reference type as a parameter. Doing so allows an object to be passed to a method. For example, consider the following program:

```
// References can be passed to methods.

using System;

class MyClass {
  int alpha, beta;
```

```
  public MyClass(int i, int j) {
    alpha = i;
    beta = j;
  }

  // Return true if ob contains the same values as the invoking object.
  public bool SameAs(MyClass ob) {
    if((ob.alpha == alpha) & (ob.beta == beta))
       return true;
    else return false;
  }

  // Make a copy of ob.
  public void Copy(MyClass ob) {
    alpha = ob.alpha;
    beta  = ob.beta;
  }

  public void Show() {
    Console.WriteLine("alpha: {0}, beta: {1}",
                      alpha, beta);
  }
}

class PassOb {
  static void Main() {
    MyClass ob1 = new MyClass(4, 5);
    MyClass ob2 = new MyClass(6, 7);

    Console.Write("ob1: ");
    ob1.Show();

    Console.Write("ob2: ");
    ob2.Show();

    if(ob1.SameAs(ob2))
      Console.WriteLine("ob1 and ob2 have the same values.");
    else
      Console.WriteLine("ob1 and ob2 have different values.");

    Console.WriteLine();

    // Now, make ob1 a copy of ob2.
    ob1.Copy(ob2);

    Console.Write("ob1 after copy: ");
    ob1.Show();

    if(ob1.SameAs(ob2))
      Console.WriteLine("ob1 and ob2 have the same values.");
    else
      Console.WriteLine("ob1 and ob2 have different values.");
  }
}
```

This program generates the following output:

```
ob1: alpha: 4, beta: 5
ob2: alpha: 6, beta: 7
ob1 and ob2 have different values.

ob1 after copy: alpha: 6, beta: 7
ob1 and ob2 have the same values.
```

The **SameAs()** and **Copy()** methods each take a reference of type **MyClass** as an argument. The **SameAs()** method compares the values of **alpha** and **beta** in the invoking object with the values of **alpha** and **beta** in the object passed via **ob**. The method returns **true** only if the two objects contain the same values for these instance variables. The **Copy()** method assigns the values of **alpha** and **beta** in the object referred to by **ob** to **alpha** and **beta** in the invoking object. As this example shows, syntactically, reference types are passed to methods in the same way as are value types.

How Arguments Are Passed

As the preceding example demonstrated, passing an object reference to a method is a straightforward task. However, there are some nuances that the example did not show. In certain cases, the effects of passing a reference type will be different than those experienced when passing a value type. To see why, let's review the two ways in which an argument can be passed to a subroutine.

The first way is *call-by-value.* This method *copies* the *value* of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument used in the call. The second way an argument can be passed is *call-by-reference.* In this method, a *reference* to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.

By default, C# uses call-by-value, which means that a copy of the argument is made and given to the receiving parameter. Thus, when you pass a value type, such as **int** or **double**, what occurs to the parameter that receives the argument has no effect outside the method. For example, consider the following program:

```
// Value types are passed by value.

using System;

class Test {
  /* This method causes no change to the arguments
     used in the call. */
  public void NoChange(int i, int j) {
    i = i + j;
    j = -j;
  }
}

class CallByValue {
  static void Main() {
    Test ob = new Test();
```

```
    int a = 15, b = 20;

    Console.WriteLine("a and b before call: " +
                      a + " " + b);

    ob.NoChange(a, b);

    Console.WriteLine("a and b after call: " +
                      a + " " + b);
  }
}
```

The output from this program is shown here:

```
a and b before call: 15 20
a and b after call: 15 20
```

As you can see, the operations that occur inside **NoChange()** have no effect on the values of **a** and **b** used in the call. Again, this is because *copies* of the *value* of **a** and **b** have been given to parameters **i** and **j**, but **a** and **b** are otherwise completely independent of **i** and **j**. Thus, assigning **i** a new value will not affect **a**.

When you pass a reference to a method, the situation is a bit more complicated. In this case, the reference, itself, is still passed by value. Thus, a copy of the reference is made and changes to the parameter will not affect the argument. (For example, making the parameter refer to a new object will not change the object to which the argument refers.) However—and this is a big however—changes *made to the object* being referred to by the parameter *will* affect the object referred to by the argument. Let's see why.

Recall that when you create a variable of a class type, you are only creating a reference to an object. Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. Therefore, the argument and the parameter will both refer to the same object. This means that objects are passed to methods by what is effectively call-by-reference. Thus, changes to the object inside the method *do* affect the object used as an argument. For example, consider the following program:

```
// Objects are passed by reference.

using System;

class Test {
  public int a, b;

  public Test(int i, int j) {
    a = i;
    b = j;
  }

  /* Pass an object. Now, ob.a and ob.b in object
     used in the call will be changed. */
  public void Change(Test ob) {
    ob.a = ob.a + ob.b;
    ob.b = -ob.b;
  }
}
```

```
class CallByRef {
  static void Main() {
    Test ob = new Test(15, 20);

    Console.WriteLine("ob.a and ob.b before call: " +
                      ob.a + " " + ob.b);

    ob.Change(ob);

    Console.WriteLine("ob.a and ob.b after call: " +
                      ob.a + " " + ob.b);
  }
}
```

This program generates the following output:

```
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 35 -20
```

As you can see, in this case, the actions inside **Change()** have affected the object used as an argument.

To review: When a reference is passed to a method, the reference itself is passed by use of call-by-value. Thus, a copy of that reference is made. However, the copy of that reference will still refer to the same object as its corresponding argument. This means that objects are implicitly passed using call-by-reference.

Use ref and out Parameters

As just explained, value types, such as **int** or **char**, are passed by value to a method. This means that changes to the parameter that receives a value type will not affect the actual argument used in the call. You can, however, alter this behavior. Through the use of the **ref** and **out** keywords, it is possible to pass any of the value types by reference. Doing so allows a method to alter the argument used in the call.

Before going into the mechanics of using **ref** and **out**, it is useful to understand why you might want to pass a value type by reference. In general, there are two reasons: to allow a method to alter the contents of its arguments or to allow a method to return more than one value. Let's look at each reason in detail.

Often you will want a method to be able to operate on the actual arguments that are passed to it. The quintessential example of this is a **Swap()** method that exchanges the values of its two arguments. Since value types are passed by value, it is not possible to write a method that swaps the value of two **int**s, for example, using C#'s default call-by-value parameter passing mechanism. The **ref** modifier solves this problem.

As you know, a **return** statement enables a method to return a value to its caller. However, a method can return *only one* value each time it is called. What if you need to return two or more pieces of information? For example, what if you want to create a method that decomposes a floating-point number into its integer and fractional parts? To do this requires that two pieces of information be returned: the integer portion and the fractional component. This method cannot be written using only a single return value. The **out** modifier solves this problem.

Use ref

The **ref** parameter modifier causes C# to create a call-by-reference, rather than a call-by-value. The **ref** modifier is specified when the method is declared and when it is called. Let's begin with a simple example. The following program creates a method called **Sqr()** that returns in-place the square of its integer argument. Notice the use and placement of **ref**.

```
// Use ref to pass a value type by reference.

using System;

class RefTest {
  // This method changes its argument. Notice the use of ref.
  public void Sqr(ref int i) {
    i = i * i;
  }
}

class RefDemo {
  static void Main() {
    RefTest ob = new RefTest();

    int a = 10;

    Console.WriteLine("a before call: " + a);

    ob.Sqr(ref a); // notice the use of ref

    Console.WriteLine("a after call: " + a);
  }
}
```

Notice that **ref** precedes the entire parameter declaration in the method and that it precedes the argument when the method is called. The output from this program, shown here, confirms that the value of the argument, **a**, was indeed modified by **Sqr()**:

```
a before call: 10
a after call: 100
```

Using **ref**, it is now possible to write a method that exchanges the values of its two value-type arguments. For example, here is a program that contains a method called **Swap()** that exchanges the values of the two integer arguments with which it is called:

```
// Swap two values.

using System;

class ValueSwap {
  // This method now changes its arguments.
  public void Swap(ref int a, ref int b) {
    int t;
```

```
    t = a;
    a = b;
    b = t;
  }
}

class ValueSwapDemo {
  static void Main() {
    ValueSwap ob = new ValueSwap();

    int x = 10, y = 20;

    Console.WriteLine("x and y before call: " + x + " " + y);

    ob.Swap(ref x, ref y);

    Console.WriteLine("x and y after call: " + x + " " + y);
  }
}
```

The output from this program is shown here:

```
x and y before call: 10 20
x and y after call: 20 10
```

Here is one important point to understand about **ref**: An argument passed by **ref** must be assigned a value prior to the call. The reason is that the method that receives such an argument assumes that the parameter refers to a valid value. Thus, using **ref**, you cannot use a method to give an argument an initial value.

Use out

Sometimes you will want to use a reference parameter to receive a value from a method, but not pass in a value. For example, you might have a method that performs some function, such as opening a network socket, that returns a success/fail code in a reference parameter. In this case, there is no information to pass into the method, but there is information to pass back out. The problem with this scenario is that a **ref** parameter must be initialized to a value prior to the call. Thus, to use a **ref** parameter would require giving the argument a dummy value just to satisfy this constraint. Fortunately, C# provides a better alternative: the **out** parameter.

An **out** parameter is similar to a **ref** parameter with this one exception: It can only be used to pass a value out of a method. It is not necessary (or useful) to give the variable used as an **out** parameter an initial value prior to calling the method. The method will give the variable a value. Furthermore, inside the method, an **out** parameter is considered *unassigned;* that is, it is assumed to have no initial value. This implies that the method *must* assign the parameter a value prior to the method's termination. Thus, after the call to the method, an **out** parameter will contain a value.

Here is an example that uses an **out** parameter. In the class **Decompose**, the **GetParts()** method decomposes a floating-point number into its integer and fractional parts. Notice how each component is returned to the caller.

```
// Use out.

using System;

class Decompose {

  /* Decompose a floating-point value into its
     integer and fractional parts. */
  public int GetParts(double n, out double frac) {
    int whole;

    whole = (int) n;
    frac = n - whole; // pass fractional part back through frac
    return whole; // return integer portion
  }
}

class UseOut {
  static void Main() {
   Decompose ob = new Decompose();
    int i;
    double f;

    i = ob.GetParts(10.125, out f);

    Console.WriteLine("Integer portion is " + i);
    Console.WriteLine("Fractional part is " + f);
  }
}
```

The output from the program is shown here:

```
Integer portion is 10
Fractional part is 0.125
```

The **GetParts()** method returns two pieces of information. First, the integer portion of **n** is returned as **GetParts()**'s return value. Second, the fractional portion of **n** is passed back to the caller through the **out** parameter **frac**. As this example shows, by using **out**, it is possible for one method to return two values.

Of course, you are not limited to only one **out** parameter. A method can return as many pieces of information as necessary through **out** parameters. Here is an example that uses two **out** parameters. The method **HasComFactor()** performs two functions. First, it determines if two integers have a common factor (other than 1). It returns **true** if they do and **false** otherwise. Second, if they do have a common factor, **HasComFactor()** returns the least and greatest common factors in **out** parameters.

```
// Use two out parameters.

using System;

class Num {
  /* Determine if x and v have a common divisor.
```

```
     If so, return least and greatest common factors in
     the out parameters. */
  public bool HasComFactor(int x, int y,
                           out int least, out int greatest) {
    int i;
    int max = x < y ? x : y;
    bool first = true;

    least = 1;
    greatest = 1;

    // Find least and greatest common factors.
    for(i=2; i <= max/2 + 1; i++) {
      if( ((y%i)==0) & ((x%i)==0) ) {
        if(first) {
          least = i;
          first = false;
        }
        greatest = i;
      }
    }

    if(least != 1) return true;
    else return false;
  }
}

class DemoOut {
  static void Main() {
    Num ob = new Num();
    int lcf, gcf;

    if(ob.HasComFactor(231, 105, out lcf, out gcf)) {
      Console.WriteLine("Lcf of 231 and 105 is " + lcf);
      Console.WriteLine("Gcf of 231 and 105 is " + gcf);
    }
    else
      Console.WriteLine("No common factor for 35 and 49.");

    if(ob.HasComFactor(35, 51, out lcf, out gcf)) {
      Console.WriteLine("Lcf of 35 and 51 " + lcf);
      Console.WriteLine("Gcf of 35 and 51 is " + gcf);
    }
    else
      Console.WriteLine("No common factor for 35 and 51.");
  }
}
```

In **Main()**, notice that **lcf** and **gcf** are not assigned values prior to the call to **HasComFactor()**. This would be an error if the parameters had been **ref** rather than **out**. The method returns either **true** or **false**, depending upon whether the two integers have a common factor. If they do, the least and greatest common factors are returned in the **out** parameters. The output from this program is shown here:

```
Lcf of 231 and 105 is 3
Gcf of 231 and 105 is 21
No common factor for 35 and 51.
```


Use ref and out on References

The use of **ref** and **out** is not limited to the passing of value types. They can also be used when a reference is passed. When **ref** or **out** modifies a reference, it causes the reference, itself, to be passed by reference. This allows a method to change the object to which the reference refers. Consider the following program, which uses **ref** reference parameters to exchange the objects to which two references are referring:

```
// Swap two references.

using System;

class RefSwap {
  int a, b;

  public RefSwap(int i, int j) {
    a = i;
    b = j;
  }

  public void Show() {
    Console.WriteLine("a: {0}, b: {1}", a, b);
  }

  // This method changes its arguments.
  public void Swap(ref RefSwap ob1, ref RefSwap ob2) {
    RefSwap t;

    t = ob1;
    ob1 = ob2;
    ob2 = t;
  }
}

class RefSwapDemo {
  static void Main() {
    RefSwap x = new RefSwap(1, 2);
    RefSwap y = new RefSwap(3, 4);

    Console.Write("x before call: ");
    x.Show();

    Console.Write("y before call: ");
    y.Show();

    Console.WriteLine();

    // Exchange the objects to which x and y refer.
    x.Swap(ref x, ref y);
```

```
    Console.Write("x after call: ");
    x.Show();

    Console.Write("y after call: ");
    y.Show();

  }
}
```

The output from this program is shown here:

```
x before call: a: 1, b: 2
y before call: a: 3, b: 4

x after call: a: 3, b: 4
y after call: a: 1, b: 2
```

In this example, the method **Swap()** exchanges the objects to which the two arguments to **Swap( )** refer. Before calling **Swap()**, **x** refers to an object that contains the values 1 and 2, and **y** refers to an object that contains the values 3 and 4. After the call to **Swap()**, **x** refers to the object that contains the values 3 and 4, and **y** refers to the object that contains the values 1 and 2. If **ref** parameters had not been used, then the exchange inside **Swap()** would have had no effect outside **Swap()**. You might want to prove this by removing **ref** from **Swap()**.

Use a Variable Number of Arguments

When you create a method, you usually know in advance the number of arguments that you will be passing to it, but this is not always the case. Sometimes you will want to create a method that can be passed an arbitrary number of arguments. For example, consider a method that finds the smallest of a set of values. Such a method might be passed as few as two values, or three, or four, and so on. In all cases, you want that method to return the smallest value. Such a method cannot be created using normal parameters. Instead, you must use a special type of parameter that stands for an arbitrary number of parameters. This is done by creating a **params** parameter.

The **params** modifier is used to declare an array parameter that will be able to receive zero or more arguments. The number of elements in the array will be equal to the number of arguments passed to the method. Your program then accesses the array to obtain the arguments.

Here is an example that uses **params** to create a method called **MinVal()**, which returns the minimum value from a set of values:

```
// Demonstrate params.

using System;

class Min {
  public int MinVal(params int[] nums) {
    int m;

    if(nums.Length == 0) {
```

```
      Console.WriteLine("Error: no arguments.");
      return 0;
    }

    m = nums[0];
    for(int i=1; i < nums.Length; i++)
      if(nums[i] < m) m = nums[i];

    return m;
  }
}

class ParamsDemo {
  static void Main() {
    Min ob = new Min();
    int min;
    int a = 10, b = 20;

    // Call with 2 values.
    min = ob.MinVal(a, b);
    Console.WriteLine("Minimum is " + min);

    // Call with 3 values.
    min = ob.MinVal(a, b, -1);
    Console.WriteLine("Minimum is " + min);

    // Call with 5 values.
    min = ob.MinVal(18, 23, 3, 14, 25);
    Console.WriteLine("Minimum is " + min);

    // Can call with an int array, too.
    int[] args = { 45, 67, 34, 9, 112, 8 };
    min = ob.MinVal(args);
    Console.WriteLine("Minimum is " + min);
  }
}
```

The output from the program is shown here:

```
Minimum is 10
Minimum is -1
Minimum is 3
Minimum is 8
```

Each time **MinVal()** is called, the arguments are passed to it via the **nums** array. The length of the array equals the number of elements. Thus, you can use **MinVal()** to find the minimum of any number of values.

Notice the last call to **MinVal()**. Rather than being passed the values individually, it is passed an array containing the values. This is perfectly legal. When a **params** parameter is created, it will accept either a variable-length list of arguments or an array containing the arguments.

Although you can pass a **params** parameter any number of arguments, they all must be of a type compatible with the array type specified by the parameter. For example, calling **MinVal()** like this:

```
min = ob.MinVal(1, 2.2); // Wrong!
```

is illegal because there is no automatic conversion from **double** (2.2) to **int**, which is the type of **nums** in **MinVal()**.

When using **params**, you need to be careful about boundary conditions because a **params** parameter can accept any number of arguments—*even zero!* For example, it is syntactically valid to call **MinVal()** as shown here:

```
min = ob.MinVal(); // no arguments
min  = ob.MinVal(3); // 1 argument
```

This is why there is a check in **MinVal()** to confirm that at least one element is in the **nums** array before there is an attempt to access that element. If the check were not there, then a runtime exception would result if **MinVal()** were called with no arguments. (Exceptions are described in Chapter 13.) Furthermore, the code in **MinVal()** was written in such a way as to permit calling **MinVal()** with one argument. In that situation, the lone argument is returned.

A method can have normal parameters and a variable-length parameter. For example, in the following program, the method **ShowArgs()** takes one **string** parameter and then a **params** integer array:

```
// Use regular parameter with a params parameter.

using System;

class MyClass {
  public void ShowArgs(string msg, params int[] nums) {
    Console.Write(msg + ": ");

    foreach(int i in nums)
      Console.Write(i + " ");

    Console.WriteLine();
  }
}

class ParamsDemo2 {
  static void Main() {
    MyClass ob = new MyClass();

    ob.ShowArgs("Here are some integers",
                1, 2, 3, 4, 5);

    ob.ShowArgs("Here are two more",
                17, 20);
  }
}
```

This program displays the following output:

```
Here are some integers: 1 2 3 4 5
Here are two more: 17 20
```

In cases where a method has regular parameters and a **params** parameter, the **params** parameter must be the last one in the parameter list. Furthermore, in all situations, there must be only one **params** parameter.

Return Objects

A method can return any type of data, including class types. For example, the following version of the **Rect** class includes a method called **Enlarge()** that creates a rectangle that is proportionally the same as the invoking rectangle, but larger by a specified factor:

```
// Return an object.

using System;

class Rect {
  int width;
  int height;

  public Rect(int w, int h) {
    width = w;
    height = h;
  }

  public int Area() {
    return width * height;
  }

  public void Show() {
    Console.WriteLine(width + " " + height);
  }

  /* Return a rectangle that is a specified
     factor larger than the invoking rectangle. */
  public Rect Enlarge(int factor) {
    return new Rect(width * factor, height * factor);
  }
}

class RetObj {
  static void Main() {
    Rect r1 = new Rect(4, 5);

    Console.Write("Dimensions of r1: ");
    r1.Show();
    Console.WriteLine("Area of r1: " + r1.Area());

    Console.WriteLine();

    // Create a rectangle that is twice as big as r1.
    Rect r2 = r1.Enlarge(2);
```

```
    Console.Write("Dimensions of r2: ");
    r2.Show();
    Console.WriteLine("Area of r2: " + r2.Area());
  }
}
```

The output is shown here:

```
Dimensions of r1: 4 5
Area of r1: 20

Dimensions of r2: 8 10
Area of r2: 80
```

When an object is returned by a method, it remains in existence until there are no more references to it. At that point, it is subject to garbage collection. Thus, an object won't be destroyed just because the method that created it terminates.

One application of object return types is the *class factory*. A class factory is a method that is used to construct objects of its class. In some situations, you may not want to give users of a class access to the class' constructor because of security concerns or because object construction depends upon certain external factors. In such cases, a class factory is used to construct objects. Here is a simple example:

```
// Use a class factory.

using System;

class MyClass {
  int a, b; // private

  // Create a class factory for MyClass.
  public MyClass Factory(int i, int j) {
    MyClass t = new MyClass();

    t.a = i;
    t.b = j;

    return t; // return an object
  }

  public void Show() {
    Console.WriteLine("a and b: " + a + " " + b);
  }

}

class MakeObjects {
  static void Main() {
    MyClass ob = new MyClass();
    int i, j;

    // Generate objects using the factory.
    for(i=0, j=10; i < 10; i++, j--) {
```

```
      MyClass anotherOb = ob.Factory(i, j); // make an object
      anotherOb.Show();
    }

    Console.WriteLine();
  }
}
```

The output is shown here:

```
a and b: 0 10
a and b: 1 9
a and b: 2 8
a and b: 3 7
a and b: 4 6
a and b: 5 5
a and b: 6 4
a and b: 7 3
a and b: 8 2
a and b: 9 1
```

Let's look closely at this example. **MyClass** does not define a constructor, so only the default constructor is available. Thus, it is not possible to set the values of **a** and **b** using a constructor. However, the class factory **Factory()** can create objects in which **a** and **b** are given values. Moreover, since **a** and **b** are private, using **Factory()** is the only way to set these values.

In **Main()**, a **MyClass** object is instantiated, and its factory method is used inside the **for** loop to create ten other objects. The line of code that creates objects is shown here:

```
MyClass anotherOb = ob.Factory(i, j); // get an object
```

With each iteration, an object reference called **anotherOb** is created, and it is assigned a reference to the object constructed by the factory. At the end of each iteration of the loop, **anotherOb** goes out of scope, and the object to which it refers is recycled.

Return an Array

Since in C# arrays are implemented as objects, a method can also return an array. (This differs from C++ in which arrays are not valid as return types.) For example, in the following program, the method **FindFactors()** returns an array that holds the factors of the argument that it is passed:

```
// Return an array.

using System;

class Factor {
  /* Return an array containing the factors of num.
     On return, numfactors will contain the number of
     factors found. */
  public int[] FindFactors(int num, out int numfactors) {
    int[] facts = new int[80]; // size of 80 is arbitrary
    int i, j;
```

```
    // Find factors and put them in the facts array.
    for(i=2, j=0; i < num/2 + 1; i++)
      if( (num%i)==0 ) {
        facts[j] = i;
        j++;
      }

    numfactors = j;
    return facts;
  }
}

class FindFactors {
  static void Main() {
    Factor f = new Factor();
    int numfactors;
    int[] factors;

    factors = f.FindFactors(1000, out numfactors);

    Console.WriteLine("Factors for 1000 are: ");
    for(int i=0; i < numfactors; i++)
      Console.Write(factors[i] + " ");

    Console.WriteLine();
  }
}
```

The output is shown here:

```
Factors for 1000 are:
2 4 5 8 10 20 25 40 50 100 125 200 250 500
```

In **Factor**, **FindFactors()** is declared like this:

```
public int[] FindFactors(int num, out int numfactors) {
```

Notice how the **int** array return type is specified. This syntax can be generalized. Whenever a method returns an array, specify it in a similar fashion, adjusting the type and dimensions as needed. For example, the following declares a method called **someMeth()** that returns a two-dimensional array of **double**:

```
public double[,] someMeth() { // ...
```

Method Overloading

In C#, two or more methods within the same class can share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be *overloaded*, and the process is referred to as *method overloading*. Method overloading is one of the ways that C# implements polymorphism.

In general, to overload a method, simply declare different versions of it. The compiler takes care of the rest. You must observe one important restriction: The type and/or number of the parameters of each overloaded method must differ. It is not sufficient for two methods to differ only in their return types. They must differ in the types or number of their parameters.

(Return types do not provide sufficient information in all cases for C# to decide which method to use.) Of course, overloaded methods *may* differ in their return types, too. When an overloaded method is called, the version of the method executed is the one whose parameters match the arguments.

Here is a simple example that illustrates method overloading:

```
// Demonstrate method overloading.

using System;

class Overload {
  public void OvlDemo() {
    Console.WriteLine("No parameters");
  }

  // Overload OvlDemo for one integer parameter.
  public void OvlDemo(int a) {
    Console.WriteLine("One parameter: " + a);
  }

  // Overload OvlDemo for two integer parameters.
  public int OvlDemo(int a, int b) {
    Console.WriteLine("Two parameters: " + a + " " + b);
    return a + b;
  }

  // Overload OvlDemo for two double parameters.
  public double OvlDemo(double a, double b) {
    Console.WriteLine("Two double parameters: " +
                      a + " "+ b);
    return a + b;
  }
}

class OverloadDemo {
  static void Main() {
    Overload ob = new Overload();
    int resI;
    double resD;

    // Call all versions of OvlDemo().
    ob.OvlDemo();
    Console.WriteLine();

    ob.OvlDemo(2);
    Console.WriteLine();

    resI = ob.OvlDemo(4, 6);
    Console.WriteLine("Result of ob.OvlDemo(4, 6): " + resI);
    Console.WriteLine();

    resD = ob.OvlDemo(1.1, 2.32);
    Console.WriteLine("Result of ob.OvlDemo(1.1, 2.32): " + resD);
  }
}
```

This program generates the following output:

```
No parameters

One parameter: 2

Two parameters: 4 6
Result of ob.OvlDemo(4, 6): 10

Two double parameters: 1.1 2.32
Result of ob.OvlDemo(1.1, 2.32): 3.42
```

As you can see, **OvlDemo()** is overloaded four times. The first version takes no parameters; the second takes one integer parameter; the third takes two integer parameters; and the fourth takes two **double** parameters. Notice that the first two versions of **OvlDemo()** return **void** and the second two return a value. This is perfectly valid, but as explained, overloading is not affected one way or the other by the return type of a method. Thus, attempting to use these two versions of **OvlDemo()** will cause an error:

```
// One OvlDemo(int) is OK.
public void OvlDemo(int a) {
  Console.WriteLine("One parameter: " + a);
}

/* Error! Two OvlDemo(int)s are not OK even though
    return types differ. */
public int OvlDemo(int a) {
  Console.WriteLine("One parameter: " + a);
  return a * a;
}
```

As the comments suggest, the difference in their return types is an insufficient difference for the purposes of overloading.

As you will recall from Chapter 3, C# provides certain implicit (i.e., automatic) type conversions. These conversions also apply to parameters of overloaded methods. For example, consider the following:

```
// Implicit type conversions can affect overloaded method resolution.

using System;

class Overload2 {
  public void MyMeth(int x) {
    Console.WriteLine("Inside MyMeth(int): " + x);
  }

  public void MyMeth(double x) {
    Console.WriteLine("Inside MyMeth(double): " + x);
  }
}

class TypeConv {
  static void Main() {
```

```
    Overload2 ob = new Overload2();

    int i = 10;
    double d = 10.1;

    byte b = 99;
    short s = 10;
    float f = 11.5F;


    ob.MyMeth(i); // calls ob.MyMeth(int)
    ob.MyMeth(d); // calls ob.MyMeth(double)

    ob.MyMeth(b); // calls ob.MyMeth(int) -- type conversion
    ob.MyMeth(s); // calls ob.MyMeth(int) -- type conversion
    ob.MyMeth(f); // calls ob.MyMeth(double) -- type conversion
  }
}
```

The output from the program is shown here:

```
Inside MyMeth(int): 10
Inside MyMeth(double): 10.1
Inside MyMeth(int): 99
Inside MyMeth(int): 10
Inside MyMeth(double): 11.5
```

In this example, only two versions of **MyMeth()** are defined: one that has an **int** parameter and one that has a **double** parameter. However, it is possible to pass **MyMeth()** a **byte**, **short**, or **float** value. In the case of **byte** and **short**, C# automatically converts them to **int**. Thus, **MyMeth(int)** is invoked. In the case of **float**, the value is converted to **double** and **MyMeth(double)** is called.

It is important to understand, however, that the implicit conversions apply only if there is no exact type match between a parameter and an argument. For example, here is the preceding program with the addition of a version of **MyMeth()** that specifies a **byte** parameter:

```
// Add MyMeth(byte).

using System;

class Overload2 {
  public void MyMeth(byte x) {
    Console.WriteLine("Inside MyMeth(byte): " + x);
  }

  public void MyMeth(int x) {
    Console.WriteLine("Inside MyMeth(int): " + x);
  }

  public void MyMeth(double x) {
    Console.WriteLine("Inside MyMeth(double): " + x);
  }
}
```

```
class TypeConv {
  static void Main() {
    Overload2 ob = new Overload2();

    int i = 10;
    double d = 10.1;

    byte b = 99;
    short s = 10;
    float f = 11.5F;


    ob.MyMeth(i); // calls ob.MyMeth(int)
    ob.MyMeth(d); // calls ob.MyMeth(double)

    ob.MyMeth(b); // calls ob.MyMeth(byte) -- now, no type conversion

    ob.MyMeth(s); // calls ob.MyMeth(int) -- type conversion
    ob.MyMeth(f); // calls ob.MyMeth(double) -- type conversion
  }
}
```

Now when the program is run, the following output is produced:

```
Inside MyMeth(int): 10
Inside MyMeth(double): 10.1
Inside MyMeth(byte): 99
Inside MyMeth(int): 10
Inside MyMeth(double): 11.5
```

In this version, since there is a version of **MyMeth()** that takes a **byte** argument, when **MyMeth()** is called with a **byte** argument, **MyMeth(byte)** is invoked and the automatic conversion to **int** does not occur.

Both **ref** and **out** participate in overload resolution. For example, the following defines two distinct and separate methods:

```
public void MyMeth(int x) {
  Console.WriteLine("Inside MyMeth(int): " + x);
}

public void MyMeth(ref int x) {
  Console.WriteLine("Inside MyMeth(ref int): " + x);
}
```

Thus,

```
ob.MyMeth(i)
```

invokes **MyMeth(int x)**, but

```
ob.MyMeth(ref i)
```

invokes **MyMeth(ref int x)**.

Although **ref** and **out** participate in overload resolution, the difference between the two alone is not sufficient. For example, these two versions of **MyMeth()** are invalid:

```
// Wrong!
public void MyMeth(out int x) { // ...
public void MyMeth(ref int x) { // ...
```

In this case, the compiler cannot differentiate between the two version of **MyMeth()** simply because one uses an **out int** parameter and the other uses a **ref int** parameter.

Method overloading supports polymorphism because it is one way that C# implements the "one interface, multiple methods" paradigm. To understand how, consider the following. In languages that do not support method overloading, each method must be given a unique name. However, frequently you will want to implement essentially the same method for different types of data. Consider the absolute value function. In languages that do not support overloading, there are usually three or more versions of this function, each with a slightly different name. For instance, in C, the function **abs()** returns the absolute value of an integer, **labs()** returns the absolute value of a long integer, and **fabs()** returns the absolute value of a floating-point value.

Since C does not support overloading, each function must have its own unique name, even though all three functions do essentially the same thing. This makes the situation more complex, conceptually, than it actually is. Although the underlying concept of each function is the same, you still have three names to remember. This situation does not occur in C# because each absolute value method can use the same name. Indeed, the .NET Framework class library includes an absolute value method called **Abs()**. This method is overloaded by the **System.Math** class to handle the numeric types. C# determines which version of **Abs()** to call based upon the type of argument.

A principal value of overloading is that it allows related methods to be accessed by use of a common name. Thus, the name **Abs** represents the *general action* that is being performed. It is left to the compiler to choose the right *specific* version for a particular circumstance. You, the programmer, need only remember the general operation being performed. Through the application of polymorphism, several names have been reduced to one. Although this example is fairly simple, if you expand the concept, you can see how overloading can help manage greater complexity.

When you overload a method, each version of that method can perform any activity you desire. There is no rule stating that overloaded methods must relate to one another. However, from a stylistic point of view, method overloading implies a relationship. Thus, while you can use the same name to overload unrelated methods, you should not. For example, you could use the name **Sqr** to create methods that return the *square* of an integer and the *square root* of a floating-point value. But these two operations are fundamentally different. Applying method overloading in this manner defeats its original purpose. In practice, you should only overload closely related operations.

C# defines the term *signature,* which is the name of a method plus its parameter list. Thus, for the purposes of overloading, no two methods within the same class can have the same signature. Notice that a signature does not include the return type since it is not used by C# for overload resolution. Also, the **params** modifier is not part of the signature.

Overload Constructors

Like methods, constructors can also be overloaded. Doing so allows you to construct objects in a variety of ways. For example, consider the following program:

```
// Demonstrate an overloaded constructor.

using System;

class MyClass {
  public int x;

  public MyClass() {
    Console.WriteLine("Inside MyClass().");
    x = 0;
  }

  public MyClass(int i) {
    Console.WriteLine("Inside MyClass(int).");
    x = i;
  }

  public MyClass(double d) {
    Console.WriteLine("Inside MyClass(double).");
    x = (int) d;
  }

  public MyClass(int i, int j) {
    Console.WriteLine("Inside MyClass(int, int).");
    x = i * j;
  }
}

class OverloadConsDemo {
  static void Main() {
    MyClass t1 = new MyClass();
    MyClass t2 = new MyClass(88);
    MyClass t3 = new MyClass(17.23);
    MyClass t4 = new MyClass(2, 4);

    Console.WriteLine("t1.x: " + t1.x);
    Console.WriteLine("t2.x: " + t2.x);
    Console.WriteLine("t3.x: " + t3.x);
    Console.WriteLine("t4.x: " + t4.x);
  }
}
```

The output from the program is shown here:

```
Inside MyClass().
Inside MyClass(int).
Inside MyClass(double).
Inside MyClass(int, int).
```

```
t1.x: 0
t2.x: 88
t3.x: 17
t4.x: 8
```

MyClass() is overloaded four ways, each constructing an object differently. The proper constructor is called based upon the arguments specified when **new** is executed. By overloading a class' constructor, you give the user of your class flexibility in the way objects are constructed.

One of the most common reasons that constructors are overloaded is to allow one object to initialize another. For example, here is an enhanced version of the **Stack** class developed earlier that allows one stack to be constructed from another:

```
// A stack class for characters.

using System;

class Stack {
  // These members are private.
  char[] stck; // holds the stack
  int tos;     // index of the top of the stack

  // Construct an empty Stack given its size.
  public Stack(int size) {
    stck = new char[size]; // allocate memory for stack
    tos = 0;
  }

  // Construct a Stack from a stack.
  public Stack(Stack ob) {
    // Allocate memory for stack.
    stck = new char[ob.stck.Length];

    // Copy elements to new stack.
    for(int i=0; i < ob.tos; i++)
      stck[i] = ob.stck[i];

    // Set tos for new stack.
    tos = ob.tos;
  }

  // Push characters onto the stack.
  public void Push(char ch) {
    if(tos==stck.Length) {
      Console.WriteLine(" -- Stack is full.");
      return;
    }

    stck[tos] = ch;
    tos++;
  }

  // Pop a character from the stack.
```

```
  public char Pop() {
    if(tos==0) {
      Console.WriteLine(" -- Stack is empty.");
      return (char) 0;
    }

    tos--;
    return stck[tos];
  }

  // Return true if the stack is full.
  public bool IsFull() {
    return tos==stck.Length;
  }

  // Return true if the stack is empty.
  public bool IsEmpty() {
    return tos==0;
  }

  // Return total capacity of the stack.
  public int Capacity() {
    return stck.Length;
  }

  // Return number of objects currently on the stack.
  public int GetNum() {
    return tos;
  }
}

// Demonstrate the Stack class.
class StackDemo {
  static void Main() {
    Stack stk1 = new Stack(10);
    char ch;
    int i;

    // Put some characters into stk1.
    Console.WriteLine("Push A through J onto stk1.");
    for(i=0; !stk1.IsFull(); i++)
      stk1.Push((char) ('A' + i));

    // Create a copy of stck1.
    Stack stk2 = new Stack(stk1);

    // Display the contents of stk1.
    Console.Write("Contents of stk1: ");
    while( !stk1.IsEmpty() ) {
      ch = stk1.Pop();
      Console.Write(ch);
    }
```

```
    Console.WriteLine();

    Console.Write("Contents of stk2: ");
    while ( !stk2.IsEmpty() ) {
      ch = stk2.Pop();
      Console.Write(ch);
    }

    Console.WriteLine("\n");

  }
}
```

The output is shown here:

```
Push A through J onto stk1.
Contents of stk1: JIHGFEDCBA
Contents of stk2: JIHGFEDCBA
```

In **StackDemo**, the first stack, **stk1**, is constructed and filled with characters. This stack is then used to construct the second stack, **stk2**. This causes the following **Stack** constructor to be executed:

```
// Construct a Stack from a stack.
public Stack(Stack ob) {
  // Allocate memory for stack.
  stck = new char[ob.stck.Length];

  // Copy elements to new stack.
  for(int i=0; i < ob.tos; i++)
    stck[i] = ob.stck[i];

  // Set tos for new stack.
  tos = ob.tos;
}
```

Inside this constructor, an array is allocated that is long enough to hold the elements contained in the stack passed in **ob**. Then, the contents of **ob**'s array are copied to the new array, and **tos** is set appropriately. After the constructor finishes, the new stack and the original stack are separate, but identical.

Invoke an Overloaded Constructor Through this

When working with overloaded constructors, it is sometimes useful for one constructor to invoke another. In C#, this is accomplished by using another form of the **this** keyword. The general form is shown here:

```
constructor-name(parameter-list1) : this(parameter-list2) {
  // ... body of constructor, which may be empty
}
```

When the constructor is executed, the overloaded constructor that matches the parameter list specified by *parameter-list2* is first executed. Then, if there are any statements inside the original constructor, they are executed. Here is an example:

```
// Demonstrate invoking a constructor through this.

using System;

class XYCoord {
  public int x, y;

  public XYCoord() : this(0, 0) {
    Console.WriteLine("Inside XYCoord()");
  }

  public XYCoord(XYCoord obj) : this(obj.x, obj.y) {
    Console.WriteLine("Inside XYCoord(obj)");
  }

  public XYCoord(int i, int j) {
    Console.WriteLine("Inside XYCoord(int, int)");
    x = i;
    y = j;
  }
}

class OverloadConsDemo {
  static void Main() {
    XYCoord t1 = new XYCoord();
    XYCoord t2 = new XYCoord(8, 9);
    XYCoord t3 = new XYCoord(t2);

    Console.WriteLine("t1.x, t1.y: " + t1.x + ", " + t1.y);
    Console.WriteLine("t2.x, t2.y: " + t2.x + ", " + t2.y);
    Console.WriteLine("t3.x, t3.y: " + t3.x + ", " + t3.y);
  }
}
```

The output from the program is shown here:

```
Inside XYCoord(int, int)
Inside XYCoord()
Inside XYCoord(int, int)
Inside XYCoord(int, int)
Inside XYCoord(obj)
t1.x, t1.y: 0, 0
t2.x, t2.y: 8, 9
t3.x, t3.y: 8, 9
```

Here is how the program works. In the **XYCoord** class, the only constructor that actually initializes the **x** and **y** fields is **XYCoord(int, int)**. The other two constructors simply invoke **XYCoord(int, int)** through **this**. For example, when object **t1** is created, its constructor, **XYCoord()**, is called. This causes **this(0, 0)** to be executed, which in this case translates into a call to **XYCoord(0, 0)**. The creation of **t2** works in similar fashion.

One reason why invoking overloaded constructors through **this** can be useful is that it can prevent the unnecessary duplication of code. In the foregoing example, there is no reason for all three constructors to duplicate the same initialization sequence, which the use of **this** avoids. Another advantage is that you can create constructors with implied "default arguments" that are used when these arguments are not explicitly specified. For example, you could create another **XYCoord** constructor as shown here:

```
public XYCoord(int x) : this(x, x) { }
```

This constructor automatically defaults the **y** coordinate to the same value as the **x** coordinate. Of course, it is wise to use such "default arguments" carefully because their misuse could easily confuse users of your classes.

Object Initializers

C# 3.0 added a new feature called *object initializers* that provides another way to create an object and initialize its fields and properties. (See Chapter 10 for a discussion of properties.) Using object initializers, you do not call a class' constructor in the normal way. Rather, you specify the names of the fields and/or properties to be initialized, giving each an initial value. Thus, the object initializer syntax provides an alternative to explicitly invoking a class' constructor. The primary use of the object initializer syntax is with anonymous types created in a LINQ expression. (Anonymous types and LINQ are described in Chapter 19.) However, because the object initializers can be used (and occasionally are used) with a named class, the fundamentals of object initialization are introduced here.

Let's begin with a simple example:

```
// A simple demonstration that uses object initializers.

using System;

class MyClass {
  public int Count;
  public string Str;
}

class ObjInitDemo {
  static void Main() {
    // Construct a MyClass object by using object initializers.
    MyClass obj = new MyClass { Count = 100, Str = "Testing" };

    Console.WriteLine(obj.Count + " " + obj.Str);
  }
}
```

This produces the following output:

```
100 Testing
```

As the output shows, the value of **obj.Count** has been initialized to 100 and the value of **obj.Str** has been initialized to "Testing". Notice, however, that **MyClass** does not define any explicit constructors, and that the normal constructor syntax has not been used. Rather, **obj** is created using the following line:

```
MyClass obj = new MyClass { Count = 100, Str = "Testing" };
```

Here, the names of the fields are explicitly specified along with their initial values. This results in a default instance of **MyClass** being constructed (by use of the implicit default constructor) and then **Count** and **Str** are given the specified initial values.

It is important to understand that the order of the initializers is not important. For example, **obj** could have been initialized as shown here:

```
MyClass obj = new MyClass { Str = "Testing", Count = 100 };
```

In this statement, the initialization of **Str** precedes the initialization of **Count**. In the program, it was the other way around. However, in either case, the end result is the same.

Here is the general form of object initialization syntax:

new *class-name* { *name* = *expr*, *name* = *expr*, *name* = *expr*, ... }

Here, *name* specifies the name of a field or property that is an accessible member of *class-name*. Of course, the type of the initializing expression specified by *expr* must be compatible with the type of field or property.

Although you can use object initializers with a named class (such as **MyClass** in the example), you usually won't. In general, you will use the normal constructor call syntax when working with named classes. As mentioned, object initializers are most applicable to anonymous types generated by a LINQ expression.

The Main( ) Method

Up to this point, you have been using one form of **Main()**. However, it has several overloaded forms. Some can be used to return a value, and some can receive arguments. Each is examined here.

Return Values from Main( )

When a program ends, you can return a value to the calling process (often the operating system) by returning a value from **Main()**. To do so, you can use this form of **Main()**:

static int Main( )

Notice that instead of being declared **void**, this version of **Main()** has a return type of **int**.

Usually, the return value from **Main()** indicates whether the program ended normally or due to some abnormal condition. By convention, a return value of zero usually indicates normal termination. All other values indicate some type of error occurred.

Pass Arguments to Main( )

Many programs accept what are called *command-line* arguments. A command-line argument is the information that directly follows the program's name on the command line when it is executed. For C# programs, these arguments are then passed to the **Main()** method. To receive the arguments, you must use one of these forms of **Main()**:

static void Main(string[ ] *args*)
static int Main(string[ ] *args*)

The first form returns **void**; the second can be used to return an integer value, as described in the preceding section. For both, the command-line arguments are stored as strings in the

string array passed to **Main()**. The length of the *args* array will be equal to the number of command-line arguments, which might be zero.

For example, the following program displays all of the command-line arguments that it is called with:

```
// Display all command-line information.

using System;

class CLDemo {
  static void Main(string[] args) {
    Console.WriteLine("There are " + args.Length +
                      " command-line arguments.");

    Console.WriteLine("They are: ");
    for(int i=0; i < args.Length; i++)
      Console.WriteLine(args[i]);
  }
}
```

If **CLDemo** is executed like this:

```
CLDemo one two three
```

you will see the following output:

```
There are 3 command-line arguments.
They are:
one
two
three
```

To understand the way that command-line arguments can be used, consider the next program. It uses a simple substitution cipher to encode or decode messages. The message to be encoded or decoded is specified on the command line. The cipher is very simple: To encode a word, each letter is incremented by 1. Thus, A becomes B, and so on. To decode, each letter is decremented. Of course, such a cipher is of no practical value, being trivially easy to break. But it does provide an enjoyable pastime for children.

```
// Encode or decode a message using a simple substitution cipher.

using System;

class Cipher {
  static int Main(string[] args) {

    // See if arguments are present.
    if(args.Length < 2) {
      Console.WriteLine("Usage: encode/decode word1 [word2...wordN]");
      return 1; // return failure code
    }

    // If args present, first arg must be encode or decode.
```

```
    if(args[0] != "encode" & args[0] != "decode") {
      Console.WriteLine("First arg must be encode or decode.");
      return 1; // return failure code
    }

    // Encode or decode message.
    for(int n=1; n < args.Length; n++) {
      for(int i=0; i < args[n].Length; i++) {
        if(args[0] == "encode")
          Console.Write((char) (args[n][i] + 1) );
        else
          Console.Write((char) (args[n][i] - 1) );
      }
      Console.Write(" ");
    }

    Console.WriteLine();

    return 0;
  }
}
```

To use the program, specify either the "encode" or "decode" command followed by the phrase that you want to encrypt or decrypt. Assuming the program is called Cipher, here are two sample runs:

```
C:>Cipher encode one two
pof uxp

C:>Cipher decode pof uxp
one two
```

There are two interesting things in this program. First, notice how the program checks that a command-line argument is present before it continues executing. This is very important and can be generalized. When a program relies on there being one or more command-line arguments, it must always confirm that the proper arguments have been supplied. Failure to do this can lead to program malfunctions. Also, since the first command-line argument must be either "encode" or "decode," the program also checks this before proceeding.

Second, notice how the program returns a termination code. If the required command line is not present, then 1 is returned, indicating abnormal termination. Otherwise, 0 is returned when the program ends.

Recursion

In C#, a method can call itself. This process is called *recursion,* and a method that calls itself is said to be *recursive.* In general, recursion is the process of defining something in terms of itself and is somewhat similar to a circular definition. The key component of a recursive method is that it contains a statement that executes a call to itself. Recursion is a powerful control mechanism.

The classic example of recursion is the computation of the factorial of a number. The factorial of a number *N* is the product of all the whole numbers between 1 and *N.* For example, 3 factorial is 1×2×3, or 6. The following program shows a recursive way to

compute the factorial of a number. For comparison purposes, a nonrecursive equivalent is also included.

```
// A simple example of recursion.

using System;

class Factorial {
  // This is a recursive method.
  public int FactR(int n) {
    int result;

    if(n==1) return 1;
    result = FactR(n-1) * n;
    return result;
  }

  // This is an iterative equivalent.
  public int FactI(int n) {
    int t, result;

    result = 1;
    for(t=1; t <= n; t++) result *= t;
    return result;
  }
}

class Recursion {
  static void Main() {
    Factorial f = new Factorial();

    Console.WriteLine("Factorials using recursive method.");
    Console.WriteLine("Factorial of 3 is " + f.FactR(3));
    Console.WriteLine("Factorial of 4 is " + f.FactR(4));
    Console.WriteLine("Factorial of 5 is " + f.FactR(5));
    Console.WriteLine();

    Console.WriteLine("Factorials using iterative method.");
    Console.WriteLine("Factorial of 3 is " + f.FactI(3));
    Console.WriteLine("Factorial of 4 is " + f.FactI(4));
    Console.WriteLine("Factorial of 5 is " + f.FactI(5));
  }
}
```

The output from this program is shown here:

```
Factorials using recursive method.
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120

Factorials using iterative method.
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

The operation of the nonrecursive method **FactI()** should be clear. It uses a loop starting at 1 and progressively multiplies each number by the moving product.

The operation of the recursive **FactR()** is a bit more complex. When **FactR()** is called with an argument of 1, the method returns 1; otherwise, it returns the product of **FactR(n–1)*n**. To evaluate this expression, **FactR()** is called with **n–1**. This process repeats until **n** equals 1 and the calls to the method begin returning. For example, when the factorial of 2 is calculated, the first call to **FactR()** will cause a second call to be made with an argument of 1. This call will return 1, which is then multiplied by 2 (the original value of **n**). The answer is then 2. You might find it interesting to insert **WriteLine()** statements into **FactR()** that show the level of recursion of each call and what the intermediate results are.

When a method calls itself, new local variables and parameters are allocated storage on the system stack, and the method code is executed with these new variables from the start. A recursive call does not make a new copy of the method. Only the arguments are new. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes at the point of the call inside the method. Recursive methods could be said to "telescope" out and back.

Here is another example of recursion. The **DisplayRev()** method uses recursion to display its string argument backward.

```
// Display a string in reverse by using recursion.

using System;

class RevStr {

  // Display a string backward.
  public void DisplayRev(string str) {
    if(str.Length > 0)
      DisplayRev(str.Substring(1, str.Length-1));
    else
      return;

    Console.Write(str[0]);
  }
}

class RevStrDemo {
  static void Main() {
    string s = "this is a test";
    RevStr rsOb = new RevStr();

    Console.WriteLine("Original string: " + s);

    Console.Write("Reversed string: ");
    rsOb.DisplayRev(s);

    Console.WriteLine();
  }
}
```

Here is the output:

```
Original string: this is a test
Reversed string: tset a si siht
```

Each time **DisplayRev()** is called, it first checks to see if **str** has a length greater than zero. If it does, it recursively calls **DisplayRev()** with a new string that consists of **str** minus its first character. This process repeats until a zero-length string is passed. This causes the recursive calls to start unraveling. As they do, the first character of **str** in each call is displayed. This results in the string being displayed in reverse order.

Recursive versions of many routines may execute a bit more slowly than the iterative equivalent because of the added overhead of the additional method calls. Too many recursive calls to a method could cause a stack overrun. Because storage for parameters and local variables is on the system stack, and each new call creates a new copy of these variables, it is possible that the stack could be exhausted. If this occurs, the CLR will throw an exception. However, you probably will not have to worry about this unless a recursive routine runs wild.

The main advantage to recursion is that some types of algorithms can be more clearly and simply implemented recursively than iteratively. For example, the quicksort sorting algorithm is quite difficult to implement in an iterative way. Also, some problems, especially AI-related ones, seem to lend themselves to recursive solutions.

When writing recursive methods, you must have a conditional statement, such as an **if**, somewhere to force the method to return without the recursive call being executed. If you don't do this, once you call the method, it will never return. This type of error is very common when working with recursion. Use **WriteLine()** statements liberally so that you can watch what is going on and abort execution if you see that you have made a mistake.

Understanding static

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally, a class member must be accessed through an object of its class, but it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword **static**. When a member is declared **static**, it can be accessed before any objects of its class are created and without reference to any object. You can declare both methods and variables to be **static**. The most common example of a **static** member is **Main()**, which is declared **static** because it must be called by the operating system when your program begins.

Outside the class, to use a **static** member, you must specify the name of its class followed by the dot operator. No object needs to be created. In fact, a **static** member cannot be accessed through an object reference. It must be accessed through its class name. For example, if you want to assign the value 10 to a **static** variable called **count** that is part of a class called **Timer**, use this line:

```
Timer.count = 10;
```

This format is similar to that used to access normal instance variables through an object, except that the class name is used. A **static** method can be called in the same way—by use of the dot operator on the name of the class.

Variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same

static variable. A **static** variable is initialized before its class is used. If no explicit initializer is specified, it is initialized to zero for numeric types, null in the case of reference types, or **false** for variables of type **bool**. Thus, a **static** variable always has a value.

The difference between a **static** method and a normal method is that the **static** method can be called through its class name, without any instance of that class being created. You have seen an example of this already: the **Sqrt()** method, which is a **static** method within C#'s **System.Math** class.

Here is an example that declares a **static** variable and a **static** method:

```
// Use static.

using System;

class StaticDemo {
  // A static variable.
  public static int Val = 100;

  // A static method.
  public static int ValDiv2() {
    return Val/2;
  }
}

class SDemo {
  static void Main() {

    Console.WriteLine("Initial value of StaticDemo.Val is "
                      + StaticDemo.Val);

    StaticDemo.Val = 8;
    Console.WriteLine("StaticDemo.Val is " + StaticDemo.Val);
    Console.WriteLine("StaticDemo.ValDiv2(): " +
                       StaticDemo.ValDiv2());
  }
}
```

The output is shown here:

```
Initial value of StaticDemo.Val is 100
StaticDemo.Val is 8
StaticDemo.ValDiv2(): 4
```

As the output shows, a **static** variable is initialized before any object of its class is created.

There are several restrictions that apply to **static** methods:

- A **static** method does not have a **this** reference. This is because a **static** method does not execute relative to any object.
- A **static** method can directly call only other **static** methods of its class. It cannot directly call an instance method of its class. The reason is that instance methods operate on specific objects, but a **static** method is not called on an object. Thus, on what object would the **static** method operate?

- A similar restriction applies to **static** data. A **static** method can directly access only other **static** data defined by its class. It cannot operate on an instance variable of its class because there is no object to operate on.

For example, in the following class, the **static** method **ValDivDenom()** is illegal:

```
class StaticError {
  public int Denom = 3; // a normal instance variable
  public static int Val = 1024; // a static variable

  /* Error! Can't directly access a non-static variable
     from within a static method. */
  static int ValDivDenom() {
    return Val/Denom; // won't compile!
  }
}
```

Here, **Denom** is a normal instance variable that cannot be accessed within a **static** method. However, the use of **Val** is okay since it is a **static** variable.

The same problem occurs when trying to call a non-**static** method from within a **static** method of the same class. For example:

```
using System;

class AnotherStaticError {
  // A non-static method.
  void NonStaticMeth() {
     Console.WriteLine("Inside NonStaticMeth().");
  }

  /* Error! Can't directly call a non-static method
     from within a static method. */
  static void staticMeth() {
    NonStaticMeth(); // won't compile
  }
}
```

In this case, the attempt to call a non-**static** (that is, instance method) from a **static** method causes a compile-time error.

It is important to understand that a **static** method *can* call instance methods and access instance variables of its class if it does so through an object of that class. It is just that it cannot use an instance variable or method without an object qualification. For example, this fragment is perfectly valid:

```
class MyClass {
  // A non-static method.
  void NonStaticMeth() {
     Console.WriteLine("Inside NonStaticMeth().");
  }

  /* Can call a non-static method through an
     object reference from within a static method. */
```

```
  public static void staticMeth(MyClass ob) {
    ob.NonStaticMeth(); // this is OK
  }
}
```

Here, **NonStaticMeth()** is called by **staticMeth()** through **ob**, which is an object of type **MyClass**.

Because **static** fields are independent of any specific object, they are useful when you need to maintain information that is applicable to an entire class. Here is an example of such a situation. It uses a **static** field to maintain a count of the number of objects that are in existence.

```
// Use a static field to count instances.

using System;

class CountInst {
  static int count = 0;

  // Increment count when object is created.
  public CountInst() {
    count++;
  }

  // Decrement count when object is destroyed.
  ~CountInst() {
    count--;
  }

  public static int GetCount() {
    return count;
  }
}

class CountDemo {
  static void Main() {
    CountInst ob;


    for(int i=0; i < 10; i++) {
      ob = new CountInst();
      Console.WriteLine("Current count: " + CountInst.GetCount());
    }
  }
}
```

The output is shown here:

```
Current count: 1
Current count: 2
Current count: 3
Current count: 4
Current count: 5
Current count: 6
```

```
Current count: 7
Current count: 8
Current count: 9
Current count: 10
```

Each time that an object of type **CountInst** is created, the **static** field **count** is incremented. Each time an object is recycled, **count** is decremented. Thus, **count** always contains a count of the number of objects currently in existence. This is possible only through the use of a **static** field. There is no way for an instance variable to maintain the count because the count relates to the class as a whole, not to a specific instance.

Here is one more example that uses **static**. Earlier in this chapter, you saw how a class factory could be used to create objects. In that example, the class factory was a non-**static** method, which meant that it could be called only through an object reference. This meant that a default object of the class needed to be created so that the factory method could be called. However, a better way to implement a class factory is as a **static** method, which allows the class factory to be called without creating an unnecessary object. Here is the class factory example rewritten to reflect this improvement:

```
// Use a static class factory.

using System;

class MyClass {
  int a, b;

  // Create a class factory for MyClass.
  static public MyClass Factory(int i, int j) {
    MyClass t = new MyClass();

    t.a = i;
    t.b = j;

    return t; // return an object
  }

  public void Show() {
    Console.WriteLine("a and b: " + a + " " + b);
  }
}

class MakeObjects {
  static void Main() {
    int i, j;

    // Generate objects using the factory.
    for(i=0, j=10; i < 10; i++, j--) {
      MyClass ob = MyClass.Factory(i, j); // get an object
      ob.Show();
    }

    Console.WriteLine();
  }
}
```

In this version, **Factory()** is invoked through its class name in this line of code:

```
MyClass ob = MyClass.Factory(i, j); // get an object
```

There is no need to create a **MyClass** object prior to using the factory.

Static Constructors

A constructor can also be specified as **static**. A **static** constructor is typically used to initialize features that apply to a class rather than an instance. Thus, it is used to initialize aspects of a class before any objects of the class are created. Here is a simple example:

```
// Use a static constructor.

using System;

class Cons {
  public static int alpha;
  public int beta;

  // A static constructor.
  static Cons() {
    alpha = 99;
    Console.WriteLine("Inside static constructor.");
  }

  // An instance constructor.
  public Cons() {
    beta = 100;
    Console.WriteLine("Inside instance constructor.");
  }
}

class ConsDemo {
  static void Main() {
    Cons ob = new Cons();

    Console.WriteLine("Cons.alpha: " + Cons.alpha);
    Console.WriteLine("ob.beta: " + ob.beta);
  }
}
```

Here is the output:

```
Inside static constructor.
Inside instance constructor.
Cons.alpha: 99
ob.beta: 100
```

Notice that the **static** constructor is called automatically (when the class is first loaded) and before the instance constructor. This can be generalized. In all cases, the **static** constructor will be executed before any instance constructor. Furthermore, **static** constructors cannot have access modifiers (thus, they use default access) and cannot be called by your program.

Static Classes

Beginning with C# 2.0, you can declare a class **static**. There are two key features of a **static** class. First, no object of a **static** class can be created. Second, a **static** class must contain only **static** members. A **static** class is created by modifying a class declaration with the keyword **static**, shown here.

static class *class-name* { // ...

Within the cla ss, all members must be explicitly specified as **static**. Making a class **static** does not automatically make its members **static**.

static classes have two primary uses. First, a **static** class is required when creating an *extension method*, which is a new feature added by C# 3.0. Extension methods relate mostly to LINQ, and a discussion of extensions methods is found in Chapter 19. Second, a **static** class is used to contain a collection of related **static** methods. This second use is demonstrated here.

The following example uses a **static** class called **NumericFn** to hold a set of **static** methods that operate on a numeric value. Because all of the members of **NumericFn** are declared **static**, the class can also be declared **static**, which prevents it from being instantiated. Thus, **NumericFn** serves an organization role, providing a good way to logically group related methods.

```
// Demonstrate a static class.

using System;

static class NumericFn {
  // Return the reciprocal of a value.
  static public double Reciprocal(double num) {
    return 1/num;
  }

  // Return the fractional part of a value.
  static public double FracPart(double num) {
    return num - (int) num;
  }

  // Return true if num is even.
  static public bool IsEven(double num) {
    return (num % 2) == 0 ? true : false;
  }

  // Return true if num is odd.
  static public bool IsOdd(double num) {
    return !IsEven(num);
  }

}

class StaticClassDemo {
  static void Main() {
    Console.WriteLine("Reciprocal of 5 is " +
                      NumericFn.Reciprocal(5.0));
```

```
    Console.WriteLine("Fractional part of 4.234 is " +
                      NumericFn.FracPart(4.234));

    if(NumericFn.IsEven(10))
      Console.WriteLine("10 is even.");

    if(NumericFn.IsOdd(5))
      Console.WriteLine("5 is odd.");

    // The following attempt to create an instance of
    // NumericFn will cause an error.
//  NumericFn ob = new NumericFn(); // Wrong!
  }
}
```

The output from the program is shown here.

```
Reciprocal of 5 is 0.2
Fractional part of 4.234 is 0.234
10 is even.
5 is odd.
```

Notice that the last line in the program is commented-out. Because **NumericFn** is a **static** class, any attempt to create an object will result in a compile-time error. It would also be an error to attempt to give **NumericFn** a non-**static** member.

One last point: Although a **static** class cannot have an instance constructor, it can have a **static** constructor.

CHAPTER 9

Operator Overloading

C# allows you to define the meaning of an operator relative to a class that you create. This process is called *operator overloading.* By overloading an operator, you expand its usage to your class. The effects of the operator are completely under your control and may differ from class to class. For example, a class that defines a linked list might use the + operator to add an object to the list. A class that implements a stack might use the + to push an object onto the stack. Another class might use the + operator in an entirely different way.

When an operator is overloaded, none of its original meaning is lost. It is simply that a new operation, relative to a specific class, is added. Therefore, overloading the + to handle a linked list, for example, does not cause its meaning relative to integers (that is, addition) to be changed.

A principal advantage of operator overloading is that it allows you to seamlessly integrate a new class type into your programming environment. This *type extensibility* is an important part of the power of an object-oriented language such as C#. Once operators are defined for a class, you can operate on objects of that class using the normal C# expression syntax. You can even use an object in expressions involving other types of data. Operator overloading is one of C#'s most powerful features.

Operator Overloading Fundamentals

Operator overloading is closely related to method overloading. To overload an operator, use the **operator** keyword to define an *operator method,* which defines the action of the operator relative to its class.

There are two forms of **operator** methods: one for unary operators and one for binary operators. The general form for each is shown here:

```
// General form for overloading a unary operator
public static ret-type operator op(param-type operand)
{
  // operations
}

// General form for overloading a binary operator
public static ret-type operator op(param-type1 operand1, param-type1 operand2)
{
  // operations
}
```

Here, the operator that you are overloading, such as **+** or **/**, is substituted for *op*. The *ret-type* specifies the type of value returned by the specified operation. Although it can be any type you choose, the return value is often of the same type as the class for which the operator is being overloaded. This correlation facilitates the use of the overloaded operator in expressions. For unary operators, the operand is passed in *operand*. For binary operators, the operands are passed in *operand1* and *operand2*. Notice that **operator** methods must be both **public** and **static**.

For unary operators, the operand must be of the same type as the class for which the operator is being defined. For binary operators, at least one of the operands must be of the same type as its class. Thus, you cannot overload any C# operators for objects that you have not created. For example, you can't redefine **+** for **int** or **string**.

One other point: Operator parameters must not use the **ref** or **out** modifier.

Overloading Binary Operators

To see how operator overloading works, let's start with an example that overloads two binary operators, the **+** and the **–**. The following program creates a class called **ThreeD**, which maintains the coordinates of an object in three-dimensional space. The overloaded **+** adds the individual coordinates of one **ThreeD** object to another. The overloaded **–** subtracts the coordinates of one object from the other.

```
// An example of operator overloading.

using System;

// A three-dimensional coordinate class.
class ThreeD {
  int x, y, z; // 3-D coordinates

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

  // Overload binary +.
  public static ThreeD operator +(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    /* This adds together the coordinates of the two points
       and returns the result. */
    result.x = op1.x + op2.x; // These are integer additions
    result.y = op1.y + op2.y; // and the + retains its original
    result.z = op1.z + op2.z; // meaning relative to them.

    return result;
  }

  // Overload binary -.
  public static ThreeD operator -(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    /* Notice the order of the operands. op1 is the left
       operand and op2 is the right. */
```

```
    result.x = op1.x - op2.x; // these are integer subtractions
    result.y = op1.y - op2.y;
    result.z = op1.z - op2.z;

    return result;
  }

  // Show X, Y, Z coordinates.
  public void Show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  static void Main() {
    ThreeD a = new ThreeD(1, 2, 3);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c;

    Console.Write("Here is a: ");
    a.Show();
    Console.WriteLine();
    Console.Write("Here is b: ");
    b.Show();
    Console.WriteLine();

    c = a + b; // add a and b together
    Console.Write("Result of a + b: ");
    c.Show();
    Console.WriteLine();

    c = a + b + c; // add a, b, and c together
    Console.Write("Result of a + b + c: ");
    c.Show();
    Console.WriteLine();

    c = c - a; // subtract a
    Console.Write("Result of c - a: ");
    c.Show();
    Console.WriteLine();

    c = c - b; // subtract b
    Console.Write("Result of c - b: ");
    c.Show();
    Console.WriteLine();
  }
}
```

This program produces the following output:

```
Here is a: 1, 2, 3

Here is b: 10, 10, 10
```

```
Result of a + b: 11, 12, 13

Result of a + b + c: 22, 24, 26

Result of c - a: 21, 22, 23

Result of c - b: 11, 12, 13
```

Let's examine the preceding program carefully, beginning with the overloaded operator **+**. When two objects of type **ThreeD** are operated on by the **+** operator, the magnitudes of their respective coordinates are added together, as shown in **operator+( )**. Notice, however, that this method does not modify the value of either operand. Instead, a new object of type **ThreeD**, which contains the result of the operation, is returned by the method. To understand why the **+** operation does not change the contents of either object, think about the standard arithmetic **+** operation as applied like this: 10 + 12. The outcome of this operation is 22, but neither 10 nor 12 is changed by it. Although no rule prevents an overloaded operator from altering the value of one of its operands, it is best for the actions of an overloaded operator to be consistent with its usual meaning.

Notice that **operator+( )** returns an object of type **ThreeD**. Although the method could have returned any valid C# type, the fact that it returns a **ThreeD** object allows the **+** operator to be used in compound expressions, such as **a+b+c**. Here, **a+b** generates a result that is of type **ThreeD**. This value can then be added to **c**. Had any other type of value been generated by **a+b**, such an expression would not work.

Here is another important point: When the coordinates are added together inside **operator+( )**, the addition of the individual coordinates results in an integer addition. This is because the individual coordinates, **x**, **y**, and **z**, are integer quantities. The fact that the **+** operator is overloaded for objects of type **ThreeD** has no effect on the **+** as it is applied to integer values.

Now, look at **operator–( )**. The **–** operator works just like the **+** operator except that the order of the parameters is important. Recall that addition is commutative, but subtraction is not. (That is, A – B is not the same as B – A!) For all binary operators, the first parameter to an operator method will contain the left operand. The second parameter will contain the one on the right. When implementing overloaded versions of the noncommutative operators, you must remember which operand is on the left and which is on the right.

Overloading Unary Operators

The unary operators are overloaded just like the binary operators. The main difference, of course, is that there is only one operand. For example, here is a method that overloads the unary minus for the **ThreeD** class:

```
// Overload unary -.
public static ThreeD operator -(ThreeD op)
{
  ThreeD result = new ThreeD();

  result.x = -op.x;
  result.y = -op.y;
  result.z = -op.z;

  return result;
}
```

Here, a new object is created that contains the negated fields of the operand. This object is then returned. Notice that the operand is unchanged. Again, this is in keeping with the usual meaning of the unary minus. For example, in an expression such as this,

```
a = -b
```

a receives the negation of **b**, but **b** is not changed.

In C#, overloading **++** and **– –** is quite easy; simply return the incremented or decremented value, but don't change the invoking object. C# will automatically handle that for you, taking into account the difference between the prefix and postfix forms. For example, here is an **operator++()** method for the **ThreeD** class:

```
// Overload unary ++.
public static ThreeD operator ++(ThreeD op)
{
  ThreeD result = new ThreeD();

  // Return the incremented result.
  result.x = op.x + 1;
  result.y = op.y + 1;
  result.z = op.z + 1;

  return result;
}
```

Here is an expanded version of the previous example program that demonstrates the unary **–** and the **++** operator:

```
// More operator overloading.

using System;

// A three-dimensional coordinate class.
class ThreeD {
  int x, y, z; // 3-D coordinates

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

  // Overload binary +.
  public static ThreeD operator +(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    /* This adds together the coordinates of the two points
       and returns the result. */
    result.x = op1.x + op2.x;
    result.y = op1.y + op2.y;
    result.z = op1.z + op2.z;

    return result;
  }

  // Overload binary -.
```

```
  public static ThreeD operator -(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    /* Notice the order of the operands. op1 is the left
       operand and op2 is the right. */
    result.x = op1.x - op2.x;
    result.y = op1.y - op2.y;
    result.z = op1.z - op2.z;

    return result;
  }

  // Overload unary -.
  public static ThreeD operator -(ThreeD op)
  {
    ThreeD result = new ThreeD();

    result.x = -op.x;
    result.y = -op.y;
    result.z = -op.z;

    return result;
  }

  // Overload unary ++.
  public static ThreeD operator ++(ThreeD op)
  {
    ThreeD result = new ThreeD();

    // Return the incremented result.
    result.x = op.x + 1;
    result.y = op.y + 1;
    result.z = op.z + 1;

    return result;
  }
  // Show X, Y, Z coordinates.
  public void Show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  static void Main() {
    ThreeD a = new ThreeD(1, 2, 3);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD();

    Console.Write("Here is a: ");
    a.Show();
    Console.WriteLine();
```

```
    Console.Write("Here is b: ");
    b.Show();
    Console.WriteLine();

    c = a + b; // add a and b together
    Console.Write("Result of a + b: ");
    c.Show();
    Console.WriteLine();

    c = a + b + c; // add a, b, and c together
    Console.Write("Result of a + b + c: ");
    c.Show();
    Console.WriteLine();

    c = c - a; // subtract a
    Console.Write("Result of c - a: ");
    c.Show();
    Console.WriteLine();

    c = c - b; // subtract b
    Console.Write("Result of c - b: ");
    c.Show();
    Console.WriteLine();

    c = -a; // assign -a to c
    Console.Write("Result of -a: ");
    c.Show();
    Console.WriteLine();

    c = a++; // post-increment a
    Console.WriteLine("Given c = a++");
    Console.Write("c is ");
    c.Show();
    Console.Write("a is ");
    a.Show();

    // Reset a to 1, 2, 3
    a = new ThreeD(1, 2, 3);
    Console.Write("\nResetting a to ");
    a.Show();

    c = ++a; // pre-increment a
    Console.WriteLine("\nGiven c = ++a");
    Console.Write("c is ");
    c.Show();
    Console.Write("a is ");
    a.Show();
  }
}
```

The output from the program is shown here:

```
Here is a: 1, 2, 3
```

```
Here is b: 10, 10, 10

Result of a + b: 11, 12, 13

Result of a + b + c: 22, 24, 26

Result of c - a: 21, 22, 23

Result of c - b: 11, 12, 13

Result of -a: -1, -2, -3

Given c = a++
c is 1, 2, 3
a is 2, 3, 4

Resetting a to 1, 2, 3

Given c = ++a
c is 2, 3, 4
a is 2, 3, 4
```

Handling Operations on C# Built-in Types

For any given class and operator, an operator method can, itself, be overloaded. One of the most common reasons for this is to allow operations between a class type and other types of data, such as a built-in type. For example, once again consider the **ThreeD** class. To this point, you have seen how to overload the **+** so that it adds the coordinates of one **ThreeD** object to another. However, this is not the only way in which you might want to define addition for **ThreeD**. For example, it might be useful to add an integer value to each coordinate of a **ThreeD** object. Such an operation could be used to translate axes. To perform such an operation, you will need to overload **+** a second time, as shown here:

```
// Overload binary + for ThreeD + int.
public static ThreeD operator +(ThreeD op1, int op2)
{
  ThreeD result = new ThreeD();

  result.x = op1.x + op2;
  result.y = op1.y + op2;
  result.z = op1.z + op2;

  return result;
}
```

Notice that the second parameter is of type **int**. Thus, the preceding method allows an integer value to be added to each field of a **ThreeD** object. This is permissible because, as explained earlier, when overloading a binary operator, one of the operands must be of the same type as the class for which the operator is being overloaded. However, the other operand can be of any other type.

Here is a version of **ThreeD** that has two overloaded **+** methods:

```
// Overload addition for ThreeD + ThreeD, and for ThreeD + int.

using System;

// A three-dimensional coordinate class.
class ThreeD {
  int x, y, z; // 3-D coordinates

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

  // Overload binary + for ThreeD + ThreeD.
  public static ThreeD operator +(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    /* This adds together the coordinates of the two points
       and returns the result. */
    result.x = op1.x + op2.x;
    result.y = op1.y + op2.y;
    result.z = op1.z + op2.z;

    return result;
  }

  // Overload binary + for object + int.
  public static ThreeD operator +(ThreeD op1, int op2)
  {
    ThreeD result = new ThreeD();

    result.x = op1.x + op2;
    result.y = op1.y + op2;
    result.z = op1.z + op2;

    return result;
  }

  // Show X, Y, Z coordinates.
  public void Show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  static void Main() {
    ThreeD a = new ThreeD(1, 2, 3);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD();

    Console.Write("Here is a: ");
    a.Show();
    Console.WriteLine();
    Console.Write("Here is b: ");
```

```
    b.Show();
    Console.WriteLine();

    c = a + b; // ThreeD + ThreeD
    Console.Write("Result of a + b: ");
    c.Show();
    Console.WriteLine();

    c = b + 10; // ThreeD + int
    Console.Write("Result of b + 10: ");
    c.Show();
  }
}
```

The output from this program is shown here:

```
Here is a: 1, 2, 3

Here is b: 10, 10, 10

Result of a + b: 11, 12, 13

Result of b + 10: 20, 20, 20
```

As the output confirms, when the **+** is applied to two **ThreeD** objects, their coordinates are added together. When the **+** is applied to a **ThreeD** object and an integer, the coordinates are increased by the integer value.

While the overloading of **+** just shown certainly adds a useful capability to the **ThreeD** class, it does not quite finish the job. Here is why. The **operator+(ThreeD, int)** method allows statements like this:

```
ob1 = ob2 + 10;
```

It does not, unfortunately, allow ones like this:

```
ob1 = 10 + ob2;
```

The reason is that the integer argument is the second argument, which is the right-hand operand, but the preceding statement puts the integer argument on the left. To allow both forms of statements, you will need to overload the **+** yet another time. This version must have its first parameter as type **int** and its second parameter as type **ThreeD**. One version of the **operator+( )** method handles **ThreeD** + integer, and the other handles integer + **ThreeD**. Overloading the **+** (or any other binary operator) this way allows a built-in type to occur on the left or right side of the operator. Here is a version **ThreeD** that overloads the **+** operator as just described:

```
// Overload the + for ThreeD + ThreeD, ThreeD + int, and int + ThreeD.

using System;

// A three-dimensional coordinate class.
class ThreeD {
  int x, y, z; // 3-D coordinates
```

```
  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

  // Overload binary + for ThreeD + ThreeD.
  public static ThreeD operator +(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    /* This adds together the coordinates of the two points
       and returns the result. */
    result.x = op1.x + op2.x;
    result.y = op1.y + op2.y;
    result.z = op1.z + op2.z;

    return result;
  }

  // Overload binary + for ThreeD + int.
  public static ThreeD operator +(ThreeD op1, int op2)
  {
    ThreeD result = new ThreeD();

    result.x = op1.x + op2;
    result.y = op1.y + op2;
    result.z = op1.z + op2;

    return result;
  }

  // Overload binary + for int + ThreeD.
  public static ThreeD operator +(int op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    result.x = op2.x + op1;
    result.y = op2.y + op1;
    result.z = op2.z + op1;

    return result;
  }

  // Show X, Y, Z coordinates.
  public void Show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  static void Main() {
    ThreeD a = new ThreeD(1, 2, 3);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD();
```

```
    Console.Write("Here is a: ");
    a.Show();
    Console.WriteLine();
    Console.Write("Here is b: ");
    b.Show();
    Console.WriteLine();

    c = a + b; // ThreeD + ThreeD
    Console.Write("Result of a + b: ");
    c.Show();
    Console.WriteLine();

    c = b + 10; // ThreeD + int
    Console.Write("Result of b + 10: ");
    c.Show();
    Console.WriteLine();

    c = 15 + b; // int + ThreeD
    Console.Write("Result of 15 + b: ");
    c.Show();
  }
}
```

The output from this program is shown here:

```
Here is a: 1, 2, 3

Here is b: 10, 10, 10

Result of a + b: 11, 12, 13

Result of b + 10: 20, 20, 20

Result of 15 + b: 25, 25, 25
```

Overloading the Relational Operators

The relational operators, such as **==** or **<**, can also be overloaded and the process is straightforward. Usually, an overloaded relational operator returns a **true** or **false** value. This is in keeping with the normal usage of these operators and allows the overloaded relational operators to be used in conditional expressions. If you return a different type result, then you are greatly restricting the operator's utility.

Here is a version of the **ThreeD** class that overloads the **<** and **>** operators. In this example, these operators compare **ThreeD** objects based on their distance from the origin. One object is greater than another if its distance from the origin is greater. One object is less than another if its distance from the origin is less than the other. Given two points, such an implementation could be used to determine which point lies on the larger sphere. If neither operator returns true, then the two points lie on the same sphere. Of course, other ordering schemes are possible.

```
// Overload < and >.

using System;

// A three-dimensional coordinate class.
class ThreeD {
  int x, y, z; // 3-D coordinates

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

  // Overload <.
  public static bool operator <(ThreeD op1, ThreeD op2)
  {
    if(Math.Sqrt(op1.x * op1.x + op1.y * op1.y + op1.z * op1.z) <
       Math.Sqrt(op2.x * op2.x + op2.y * op2.y + op2.z * op2.z))
      return true;
    else
      return false;
  }

  // Overload >.
  public static bool operator >(ThreeD op1, ThreeD op2)
  {
    if(Math.Sqrt(op1.x * op1.x + op1.y * op1.y + op1.z * op1.z) >
       Math.Sqrt(op2.x * op2.x + op2.y * op2.y + op2.z * op2.z))
      return true;
    else
      return false;
  }

  // Show X, Y, Z coordinates.
  public void Show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  static void Main() {
    ThreeD a = new ThreeD(5, 6, 7);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD(1, 2, 3);
    ThreeD d = new ThreeD(6, 7, 5);

    Console.Write("Here is a: ");
    a.Show();
    Console.Write("Here is b: ");
    b.Show();
    Console.Write("Here is c: ");
    c.Show();
    Console.Write("Here is d: ");
    d.Show();
```

```
    Console.WriteLine();

    if(a > c) Console.WriteLine("a > c is true");
    if(a < c) Console.WriteLine("a < c is true");
    if(a > b) Console.WriteLine("a > b is true");
    if(a < b) Console.WriteLine("a < b is true");

    if(a > d) Console.WriteLine("a > d is true");
    else if(a < d) Console.WriteLine("a < d is true");
    else Console.WriteLine("a and d are same distance from origin");
  }
}
```

The output from this program is shown here:

```
Here is a: 5, 6, 7
Here is b: 10, 10, 10
Here is c: 1, 2, 3
Here is d: 6, 7, 5

a > c is true
a < b is true
a and d are same distance from origin
```

An important restriction applies to overloading the relational operators: You must overload them in pairs. For example, if you overload **<**, you must also overload **>**, and vice versa. The operator pairs are

= =	!=
<	>
<=	>=

One other point: If you overload the **= =** and **!=** operators, then you will usually need to override **Object.Equals( )** and **Object.GetHashCode( )**. These methods and the technique of overriding are discussed in Chapter 11.

Overloading true and false

The keywords **true** and **false** can also be used as unary operators for the purposes of overloading. Overloaded versions of these operators provide custom determinations of true and false relative to classes that you create. Once true and false are overloaded for a class, you can use objects of that class to control the **if**, **while**, **for**, and **do-while** statements, or in a **?** expression.

The **true** and **false** operators must be overloaded as a pair. You cannot overload just one. Both are unary operators and they have this general form:

```
public static bool operator true(param-type operand)
{
  // return true or false
}
```

```
public static bool operator false(param-type operand)
{
  // return true or false
}
```

Notice that each returns a **bool** result.

The following example shows how **true** and **false** can be implemented for the **ThreeD** class. Each assumes that a **ThreeD** object is true if at least one coordinate is non-zero. If all three coordinates are zero, then the object is false. The decrement operator is also implemented for the purpose of illustration.

```
// Overload true and false for ThreeD.

using System;

// A three-dimensional coordinate class.
class ThreeD {
  int x, y, z; // 3-D coordinates

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

  // Overload true.
  public static bool operator true(ThreeD op) {
    if((op.x != 0) || (op.y != 0) || (op.z != 0))
      return true; // at least one coordinate is non-zero
    else
      return false;
  }

  // Overload false.
  public static bool operator false(ThreeD op) {
    if((op.x == 0) && (op.y == 0) && (op.z == 0))
      return true; // all coordinates are zero
    else
      return false;
  }

  // Overload unary --.
  public static ThreeD operator --(ThreeD op)
  {
    ThreeD result = new ThreeD();

    // Return the decremented result.
    result.x = op.x - 1;
    result.y = op.y - 1;
    result.z = op.z - 1;

    return result;
  }

  // Show X, Y, Z coordinates.
  public void Show()
  {
```

```
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class TrueFalseDemo {
  static void Main() {
    ThreeD a = new ThreeD(5, 6, 7);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD(0, 0, 0);

    Console.Write("Here is a: ");
    a.Show();
    Console.Write("Here is b: ");
    b.Show();
    Console.Write("Here is c: ");
    c.Show();
    Console.WriteLine();

    if(a) Console.WriteLine("a is true.");
    else Console.WriteLine("a is false.");

    if(b) Console.WriteLine("b is true.");
    else Console.WriteLine("b is false.");

    if(c) Console.WriteLine("c is true.");
    else Console.WriteLine("c is false.");

    Console.WriteLine();

    Console.WriteLine("Control a loop using a ThreeD object.");
    do {
      b.Show();
      b--;
    } while(b);
  }
}
```

The output is shown here:

```
Here is a: 5, 6, 7
Here is b: 10, 10, 10
Here is c: 0, 0, 0

a is true.
b is true.
c is false.

Control a loop using a ThreeD object.
10, 10, 10
9, 9, 9
8, 8, 8
7, 7, 7
6, 6, 6
5, 5, 5
```

```
4, 4, 4
3, 3, 3
2, 2, 2
1, 1, 1
```

Notice how the **ThreeD** objects are used to control **if** statements and a **while** loop. In the case of the **if** statements, the **ThreeD** object is evaluated using **true**. If the result of this operation is true, then the **if** statement succeeds. In the case of the **do-while** loop, each iteration of the loop decrements **b**. The loop repeats as long as **b** evaluates as true (that is, it contains at least one non-zero coordinate). When **b** contains all zero coordinates, it evaluates as false when the **true** operator is applied and the loop stops.

Overloading the Logical Operators

As you know, C# defines the following logical operators: **&**, **|**, **!**, **&&**, and **||**. Of these, only the **&**, **|**, and **!** can be overloaded. By following certain rules, however, the benefits of the short-circuit **&&** and **||** can still be obtained. Each situation is examined here.

A Simple Approach to Overloading the Logical Operators

Let's begin with the simplest situation. If you will not be making use of the short-circuit logical operators, then you can overload **&** and **|** as you would intuitively think, with each returning a **bool** result. An overloaded **!** will also usually return a **bool** result.

Here is an example that overloads the **!**, **&**, and **|** logical operators for objects of type **ThreeD**. As before, each assumes that a **ThreeD** object is true if at least one coordinate is non-zero. If all three coordinates are zero, then the object is false.

```
// A simple way to overload !, |, and & for ThreeD.

using System;

// A three-dimensional coordinate class.
class ThreeD {
  int x, y, z; // 3-D coordinates

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }


  // Overload |.
  public static bool operator |(ThreeD op1, ThreeD op2)
  {
    if( ((op1.x != 0) || (op1.y != 0) || (op1.z != 0)) |
        ((op2.x != 0) || (op2.y != 0) || (op2.z != 0)) )
      return true;
    else
      return false;
  }

  // Overload &.
  public static bool operator &(ThreeD op1, ThreeD op2)
  {
```

```
    if( ((op1.x != 0) && (op1.y != 0) && (op1.z != 0)) &
       ((op2.x != 0) && (op2.y != 0) && (op2.z != 0)) )
      return true;
    else
      return false;
  }

  // Overload !.
  public static bool operator !(ThreeD op)
  {
    if((op.x != 0) || (op.y != 0) || (op.z != 0))
      return false;
    else return true;
  }

  // Show X, Y, Z coordinates.
  public void Show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class TrueFalseDemo {
  static void Main() {
    ThreeD a = new ThreeD(5, 6, 7);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD(0, 0, 0);

    Console.Write("Here is a: ");
    a.Show();
    Console.Write("Here is b: ");
    b.Show();
    Console.Write("Here is c: ");
    c.Show();
    Console.WriteLine();

    if(!a) Console.WriteLine("a is false.");
    if(!b) Console.WriteLine("b is false.");
    if(!c) Console.WriteLine("c is false.");

    Console.WriteLine();

    if(a & b) Console.WriteLine("a & b is true.");
    else Console.WriteLine("a & b is false.");

    if(a & c) Console.WriteLine("a & c is true.");
    else Console.WriteLine("a & c is false.");

    if(a | b) Console.WriteLine("a | b is true.");
    else Console.WriteLine("a | b is false.");

    if(a | c) Console.WriteLine("a | c is true.");
    else Console.WriteLine("a | c is false.");
  }
}
```

The output from the program is shown here:

```
Here is a: 5, 6, 7
Here is b: 10, 10, 10
Here is c: 0, 0, 0

c is false.

a & b is true.
a & c is false.
a | b is true.
a | c is true.
```

In this approach, the **&**, **|**, and **!** operator methods each return a **bool** result. This is necessary if the operators are to be used in their normal manner (that is, in places that expect a **bool** result). Recall that for all built-in types, the outcome of a logical operation is a value of type **bool**. Thus, having the overloaded versions of these operators return type **bool** is a rational approach. Unfortunately, this approach works only if you will not be needing the short-circuit operators.

Enabling the Short-Circuit Operators

To enable the use of the **&&** and **||** short-circuit operators, you must follow four rules. First, the class must overload **&** and **|**. Second, the return type of the overloaded **&** and **|** methods must be the same as the class for which the operators are being overloaded. Third, each parameter must be a reference to an object of the class for which the operator is being overloaded. Fourth, the **true** and **false** operators must be overloaded for the class. When these conditions have been met, the short-circuit operators automatically become available for use.

The following program shows how to properly implement the **&** and **|** for the **ThreeD** class so that the short-circuit operators **&&** and **||** are available.

```
/* A better way to overload !, |, and & for ThreeD.
   This version automatically enables the && and || operators. */

using System;

// A three-dimensional coordinate class.
class ThreeD {
  int x, y, z; // 3-D coordinates

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

  // Overload | for short-circuit evaluation.
  public static ThreeD operator |(ThreeD op1, ThreeD op2)
  {
    if( ((op1.x != 0) || (op1.y != 0) || (op1.z != 0)) |
        ((op2.x != 0) || (op2.y != 0) || (op2.z != 0)) )
      return new ThreeD(1, 1, 1);
    else
      return new ThreeD(0, 0, 0);
  }
```

```
  // Overload & for short-circuit evaluation.
  public static ThreeD operator &(ThreeD op1, ThreeD op2)
  {
    if( ((op1.x != 0) && (op1.y != 0) && (op1.z != 0)) &
       ((op2.x != 0) && (op2.y != 0) && (op2.z != 0)) )
      return new ThreeD(1, 1, 1);
    else
      return new ThreeD(0, 0, 0);
  }

  // Overload !.
  public static bool operator !(ThreeD op)
  {
    if(op) return false;
    else return true;
  }

  // Overload true.
  public static bool operator true(ThreeD op) {
    if((op.x != 0) || (op.y != 0) || (op.z != 0))
      return true; // at least one coordinate is non-zero
    else
      return false;
  }

  // Overload false.
  public static bool operator false(ThreeD op) {
    if((op.x == 0) && (op.y == 0) && (op.z == 0))
      return true; // all coordinates are zero
    else
      return false;
  }

  // Show X, Y, Z coordinates.
  public void Show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class TrueFalseDemo {
  static void Main() {
    ThreeD a = new ThreeD(5, 6, 7);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD(0, 0, 0);

    Console.Write("Here is a: ");
    a.Show();
    Console.Write("Here is b: ");
    b.Show();
    Console.Write("Here is c: ");
    c.Show();
    Console.WriteLine();
```

```
    if(a) Console.WriteLine("a is true.");
    if(b) Console.WriteLine("b is true.");
    if(c) Console.WriteLine("c is true.");

    if(!a) Console.WriteLine("a is false.");
    if(!b) Console.WriteLine("b is false.");
    if(!c) Console.WriteLine("c is false.");

    Console.WriteLine();

    Console.WriteLine("Use & and |");
    if(a & b) Console.WriteLine("a & b is true.");
    else Console.WriteLine("a & b is false.");

    if(a & c) Console.WriteLine("a & c is true.");
    else Console.WriteLine("a & c is false.");

    if(a | b) Console.WriteLine("a | b is true.");
    else Console.WriteLine("a | b is false.");

    if(a | c) Console.WriteLine("a | c is true.");
    else Console.WriteLine("a | c is false.");

    Console.WriteLine();

    // Now use short-circuit ops.
    Console.WriteLine("Use short-circuit && and ||");
    if(a && b) Console.WriteLine("a && b is true.");
    else Console.WriteLine("a && b is false.");

    if(a && c) Console.WriteLine("a && c is true.");
    else Console.WriteLine("a && c is false.");

    if(a || b) Console.WriteLine("a || b is true.");
    else Console.WriteLine("a || b is false.");

    if(a || c) Console.WriteLine("a || c is true.");
    else Console.WriteLine("a || c is false.");
  }
}
```

The output from the program is shown here:

```
Here is a: 5, 6, 7
Here is b: 10, 10, 10
Here is c: 0, 0, 0

a is true.
b is true.
c is false.

Use & and |
a & b is true.
```

```
a & c is false.
a | b is true.
a | c is true.

Use short-circuit && and ||
a && b is true.
a && c is false.
a || b is true.
a || c is true.
```

Let's look closely at how the **&** and **|** are implemented. They are shown here:

```
// Overload | for short-circuit evaluation.
public static ThreeD operator |(ThreeD op1, ThreeD op2)
{
  if( ((op1.x != 0) || (op1.y != 0) || (op1.z != 0)) |
     ((op2.x != 0) || (op2.y != 0) || (op2.z != 0)) )
    return new ThreeD(1, 1, 1);
  else
    return new ThreeD(0, 0, 0);
}

// Overload & for short-circuit evaluation.
public static ThreeD operator &(ThreeD op1, ThreeD op2)
{
  if( ((op1.x != 0) && (op1.y != 0) && (op1.z != 0)) &
     ((op2.x != 0) && (op2.y != 0) && (op2.z != 0)) )
    return new ThreeD(1, 1, 1);
  else
    return new ThreeD(0, 0, 0);
}
```

Notice first that both now return an object of type **ThreeD**. Pay attention to how this object is generated. If the outcome of the operation is true, then a true **ThreeD** object (one in which at least one coordinate is non-zero) is created and returned. If the outcome is false, then a false object is created and returned. Thus, in a statement like this

```
if(a & b) Console.WriteLine("a & b is true.");
else Console.WriteLine("a & b is false.");
```

the outcome of **a & b** is a **ThreeD** object, which in this case is a true object. Since the operators **true** and **false** are defined, this resulting object is subjected to the **true** operator, and a **bool** result is returned. In this case, the result is **true** and the **if** succeeds.

Because the necessary rules have been followed, the short-circuit operators are now available for use on **ThreeD** objects. They work like this. The first operand is tested by using **operator true** (for **||**) or **operator false** (for **&&**). If it can determine the outcome of the operation, then the corresponding **&** or **|** is not evaluated. Otherwise, the corresponding overloaded **&** or **|** is used to determine the result. Thus, using a **&&** or **||** causes the corresponding **&** or **|** to be invoked only when the first operand cannot determine the outcome of the expression. For example, consider this statement from the program:

```
if(a || c) Console.WriteLine("a || c is true.");
```

The **true** operator is first applied to **a**. Since **a** is true in this situation, there is no need to use the | operator method. However, if the statement were rewritten like this:

```
if(c || a) Console.WriteLine("c || a is true.");
```

then the **true** operator would first be applied to **c**, which in this case is false. Thus, the | operator method would be invoked to determine if **a** was true (which it is in this case).

Although you might at first think that the technique used to enable the short-circuit operators is a bit convoluted, it makes sense if you think about it a bit. By overloading **true** and **false** for a class, you enable the compiler to utilize the short-circuit operators without having to explicitly overload either. Furthermore, you gain the ability to use objects in conditional expressions. In general, unless you need a very narrow implementation of **&** and |, you are better off creating a full implementation.

Conversion Operators

In some situations, you will want to use an object of a class in an expression involving other types of data. Sometimes, overloading one or more operators can provide the means of doing this. However, in other cases, what you want is a simple type conversion from the class type to the target type. To handle these cases, C# allows you to create a special type of **operator** method called a *conversion operator.* A conversion operator converts an object of your class into another type. Conversion operators help fully integrate class types into the C# programming environment by allowing objects of a class to be freely mixed with other data types as long as a conversion to those other types is defined.

There are two forms of conversion operators, implicit and explicit. The general form for each is shown here:

public static operator implicit *target-type*(*source-type v*) { return *value*; }
public static operator explicit *target-type*(*source-type v*) { return *value*; }

Here, *target-type* is the target type that you are converting to; *source-type* is the type you are converting from; and *value* is the value of the class after conversion. The conversion operators return data of type *target-type*, and no other return type specifier is allowed.

If the conversion operator specifies **implicit**, then the conversion is invoked automatically, such as when an object is used in an expression with the target type. When the conversion operator specifies **explicit**, the conversion is invoked when a cast is used. You cannot define both an implicit and explicit conversion operator for the same target and source types.

To illustrate a conversion operator, we will create one for the **ThreeD** class. Suppose you want to convert an object of type **ThreeD** into an integer so it can be used in an integer expression. Further, the conversion will take place by using the product of the three dimensions. To accomplish this, you will use an implicit conversion operator that looks like this:

```
public static implicit operator int(ThreeD op1)
{
  return op1.x * op1.y * op1.z;
}
```

Here is a program that illustrates this conversion operator:

```
// An example that uses an implicit conversion operator.

using System;

// A three-dimensional coordinate class.
class ThreeD {
  int x, y, z; // 3-D coordinates

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

  // Overload binary +.
  public static ThreeD operator +(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    result.x = op1.x + op2.x;
    result.y = op1.y + op2.y;
    result.z = op1.z + op2.z;

    return result;
  }

  // An implicit conversion from ThreeD to int.
  public static implicit operator int(ThreeD op1)
  {
    return op1.x * op1.y * op1.z;
  }

  // Show X, Y, Z coordinates.
  public void Show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  static void Main() {
    ThreeD a = new ThreeD(1, 2, 3);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD();
    int i;

    Console.Write("Here is a: ");
    a.Show();
    Console.WriteLine();
    Console.Write("Here is b: ");
    b.Show();
    Console.WriteLine();

    c = a + b; // add a and b together
    Console.Write("Result of a + b: ");
```

```
    c.Show();
    Console.WriteLine();

    i = a; // convert to int
    Console.WriteLine("Result of i = a: " + i);
    Console.WriteLine();

    i = a * 2 - b; // convert to int
    Console.WriteLine("result of a * 2 - b: " + i);
  }
}
```

This program displays the output:

```
Here is a: 1, 2, 3

Here is b: 10, 10, 10

Result of a + b: 11, 12, 13

Result of i = a: 6

result of a * 2 - b: -988
```

As the program illustrates, when a **ThreeD** object is used in an integer expression, such as **i = a**, the conversion is applied to the object. In this specific case, the conversion returns the value 6, which is the product of coordinates stored in **a**. However, when an expression does not require a conversion to **int**, the conversion operator is not called. This is why **c = a + b** *does not* invoke **operator int( )**.

Remember that you can create different conversion operators to meet different needs. You could define a second conversion operator that converts **ThreeD** to **double**, for example. Each conversion is applied automatically and independently.

An implicit conversion operator is applied automatically when a conversion is required in an expression, when passing an object to a method, in an assignment, and also when an explicit cast to the target type is used. Alternatively, you can create an explicit conversion operator, which is invoked only when an explicit cast is used. An explicit conversion operator is not invoked automatically. For example, here is the previous program reworked to use an explicit conversion to **int**:

```
// Use an explicit conversion.

using System;

// A three-dimensional coordinate class.
class ThreeD {
  int x, y, z; // 3-D coordinates

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

  // Overload binary +.
  public static ThreeD operator +(ThreeD op1, ThreeD op2)
  {
```

```
    ThreeD result = new ThreeD();

    result.x = op1.x + op2.x;
    result.y = op1.y + op2.y;
    result.z = op1.z + op2.z;

    return result;
  }

  // This is now explicit.
  public static explicit operator int(ThreeD op1)
  {
    return op1.x * op1.y * op1.z;
  }

  // Show X, Y, Z coordinates.
  public void Show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  static void Main() {
    ThreeD a = new ThreeD(1, 2, 3);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD();
    int i;

    Console.Write("Here is a: ");
    a.Show();
    Console.WriteLine();
    Console.Write("Here is b: ");
    b.Show();
    Console.WriteLine();

    c = a + b; // add a and b together
    Console.Write("Result of a + b: ");
    c.Show();
    Console.WriteLine();

    i = (int) a; // explicitly convert to int -- cast required
    Console.WriteLine("Result of i = a: " + i);
    Console.WriteLine();

    i = (int)a * 2 - (int)b; // casts required
    Console.WriteLine("result of a * 2 - b: " + i);

  }
}
```

Because the conversion operator is now marked as explicit, conversion to **int** must be explicitly cast. For example, in this line:

```
i = (int) a; // explicitly convert to int -- cast required
```

if you remove the cast, the program will not compile.

There are a few restrictions to conversion operators:

- Either the target type or the source type of the conversion must be the class in which the conversion is declared. You cannot, for example, redefine the conversion from **double** to **int**.
- You cannot define a conversion to or from **object**.
- You cannot define both an implicit and an explicit conversion for the same source and target types.
- You cannot define a conversion from a base class to a derived class. (See Chapter 11 for a discussion of base and derived classes.)
- You cannot define a conversion from or to an interface. (See Chapter 12 for a discussion of interfaces.)

In addition to these rules, there are suggestions that you should normally follow when choosing between implicit and explicit conversion operators. Although convenient, implicit conversions should be used only in situations in which the conversion is inherently error-free. To ensure this, implicit conversions should be created only when these two conditions are met: First, that no loss of information, such as truncation, overflow, or loss of sign, occurs. Second, that the conversion does not cause an exception. If the conversion cannot meet these two requirements, then you should use an explicit conversion.

Operator Overloading Tips and Restrictions

The action of an overloaded operator as applied to the class for which it is defined need not bear any relationship to that operator's default usage, as applied to C#'s built-in types. However, for the purposes of the structure and readability of your code, an overloaded operator should reflect, when possible, the spirit of the operator's original use. For example, the **+** relative to **ThreeD** is conceptually similar to the **+** relative to integer types. There would be little benefit in defining the **+** operator relative to some class in such a way that it acts more the way you would expect the **/** operator to perform, for instance. The central concept is that while you can give an overloaded operator any meaning you like, for clarity it is best when its new meaning is related to its original meaning.

There are some restrictions to overloading operators. You cannot alter the precedence of any operator. You cannot alter the number of operands required by the operator, although your operator method could choose to ignore an operand. There are several operators that you cannot overload. Perhaps most significantly, you cannot overload any assignment operator, including the compound assignments, such as **+=**. Here are the other operators that cannot be overloaded. (This list includes several operators that are discussed later in this book.)

&&	()	.	?
??	[ ]	||	=
=>	–>	as	checked
default	is	new	sizeof
typeof	unchecked		

Although you cannot overload the cast operator **( )** explicitly, you can create conversion operators, as shown earlier, that perform this function.

It may seem like a serious restriction that operators such as **+=** can't be overloaded, but it isn't. In general, if you have defined an operator, then if that operator is used in a compound assignment, your overloaded operator method is invoked. Thus, **+=** automatically uses your version of **operator+( )**. For example, assuming the **ThreeD** class, if you use a sequence like this

```
ThreeD a = new ThreeD(1, 2, 3);
ThreeD b = new ThreeD(10, 10, 10);

b += a; // add a and b together
```

ThreeD's **operator+( )** is automatically invoked, and **b** will contain the coordinate 11, 12, 13.

One last point: Although you cannot overload the **[ ]** array indexing operator using an **operator** method, you can create indexers, which are described in the next chapter.

Another Example of Operator Overloading

Throughout this chapter we have been using the **ThreeD** class to demonstrate operator overloading, and in this regard it has served us well. Before concluding this chapter, however, it is useful to work through another example. Although the general principles of operator overloading are the same no matter what class is used, the following example helps show the power of operator overloading—especially where type extensibility is concerned.

This example develops a four-bit integer type and defines several operations for it. As you might know, in the early days of computing, the four-bit quantity was common because it represented half a byte. It is also large enough to hold one hexadecimal digit. Since four bits are half a byte, a four-bit quantity is sometimes referred to as a *nybble.* In the days of front-panel machines in which programmers entered code one nybble at a time, thinking in terms of nybbles was an everyday affair! Although not as common now, a four-bit type still makes an interesting addition to the other C# integers. Traditionally, a nybble is an unsigned value.

The following example uses the **Nybble** class to implement a nybble data type. It uses an **int** for its underlying storage, but it restricts the values that can be held to 0 through 15. It defines the following operators:

- Addition of a **Nybble** to a **Nybble**
- Addition of an **int** to a **Nybble**
- Addition of a **Nybble** to an **int**
- Greater than and less than
- The increment operator
- Conversion to **Nybble** from **int**
- Conversion to **int** from **Nybble**

These operations are sufficient to show how a class type can be fully integrated into the C# type system. However, for complete **Nybble** implementation, you will need to define all of the other operators. You might want to try adding others on your own.

The complete **Nybble** class is shown here along with a **NybbleDemo**, which demonstrates its use:

```
// Create a 4-bit type called Nybble.

using System;

// A 4-bit type.
class Nybble {
  int val; // underlying storage

  public Nybble() { val = 0; }

  public Nybble(int i) {
    val = i;
    val = val & 0xF; // retain lower 4 bits
  }

  // Overload binary + for Nybble + Nybble.
  public static Nybble operator +(Nybble op1, Nybble op2)
  {
    Nybble result = new Nybble();

    result.val = op1.val + op2.val;

    result.val = result.val & 0xF; // retain lower 4 bits

    return result;
  }

  // Overload binary + for Nybble + int.
  public static Nybble operator +(Nybble op1, int op2)
  {
    Nybble result = new Nybble();

    result.val = op1.val + op2;

    result.val = result.val & 0xF; // retain lower 4 bits

    return result;
  }

  // Overload binary + for int + Nybble.
  public static Nybble operator +(int op1, Nybble op2)
  {
    Nybble result = new Nybble();

    result.val = op1 + op2.val;

    result.val = result.val & 0xF; // retain lower 4 bits

    return result;
  }

  // Overload ++.
```

```
  public static Nybble operator ++(Nybble op)
  {
    Nybble result = new Nybble();
    result.val = op.val + 1;

    result.val = result.val & 0xF; // retain lower 4 bits

    return result;
  }

  // Overload >.
  public static bool operator >(Nybble op1, Nybble op2)
  {
    if(op1.val > op2.val) return true;
    else return false;
  }

  // Overload <.
  public static bool operator <(Nybble op1, Nybble op2)
  {
    if(op1.val < op2.val) return true;
    else return false;
  }

  // Convert a Nybble into an int.
  public static implicit operator int (Nybble op)
  {
    return op.val;
  }

  // Convert an int into a Nybble.
  public static implicit operator Nybble (int op)
  {
    return new Nybble(op);
  }
}

class NybbleDemo {
  static void Main() {
    Nybble a = new Nybble(1);
    Nybble b = new Nybble(10);
    Nybble c = new Nybble();
    int t;

    Console.WriteLine("a: " + (int) a);
    Console.WriteLine("b: " + (int) b);

    // Use a Nybble in an if statement.
    if(a < b) Console.WriteLine("a is less than b\n");

    // Add two Nybbles together.
    c = a + b;
    Console.WriteLine("c after c = a + b: " + (int) c);
```

```
    // Add an int to a Nybble.
    a += 5;
    Console.WriteLine("a after a += 5: " + (int) a);

    Console.WriteLine();

    // Use a Nybble in an int expression.
    t = a * 2 + 3;
    Console.WriteLine("Result of a * 2 + 3: " + t);

    Console.WriteLine();

    // Illustrate int assignment and overflow.
    a = 19;
    Console.WriteLine("Result of a = 19: " + (int) a);

    Console.WriteLine();

    // Use a Nybble to control a loop.
    Console.WriteLine("Control a for loop with a Nybble.");
    for(a = 0; a < 10; a++)
      Console.Write((int) a + " ");

    Console.WriteLine();
  }
}
```

The output from the program is shown here:

```
a: 1
b: 10
a is less than b

c after c = a + b: 11
a after a += 5: 6

Result of a * 2 + 3: 15

Result of a = 19: 3

Control a for loop with a Nybble.
0 1 2 3 4 5 6 7 8 9
```

Although most of the operation of **Nybble** should be easy to understand, there is one important point to make: The conversion operators play a large role in the integration of **Nybble** into the C# type system. Because conversions are defined from **Nybble** to **int** and from **int** to **Nybble**, a **Nybble** object can be freely mixed in arithmetic expressions. For example, consider this expression from the program:

```
t = a * 2 + 3;
```

Here, **t** is an **int**, as are 2 and 3, but **a** is a **Nybble**. These two types are compatible in the expression because of the implicit conversion of **Nybble** to **int**. In this case, since the rest of the expression is of type **int**, **a** is converted to **int** by its conversion method.

The conversion from **int** to **Nybble** allows a **Nybble** object to be assigned an **int** value. For example, in the program, the statement

```
a = 19;
```

works like this. The conversion operator from **int** to **Nybble** is executed. This causes a new **Nybble** object to be created that contains the low-order 4 bits of the value 19 (which is 3 because 19 overflows the range of a **Nybble**). This object is then assigned to **a**. Without the conversion operators, such expressions would not be allowed.

The conversion of **Nybble** to **int** is also used by the **for** loop. Without this conversion, it would not be possible to write the **for** loop in such a straightforward way.

CHAPTER 10

Indexers and Properties

This chapter examines two special types of class members that have a close relationship to each other: indexers and properties. Each expands the power of a class by enhancing its integration into C#'s type system and improving its resiliency. Indexers provide the mechanism by which an object can be indexed like an array. Properties offer a streamlined way to manage access to a class' instance data. They relate to each other because both rely upon another feature of C#: the accessor.

Indexers

As you know, array indexing is performed using the [ ] operator. It is possible to define the [ ] operator for classes that you create, but you don't use an **operator** method. Instead, you create an *indexer.* An indexer allows an object to be indexed like an array. The main use of indexers is to support the creation of specialized arrays that are subject to one or more constraints. However, you can use an indexer for any purpose for which an array-like syntax is beneficial. Indexers can have one or more dimensions. We will begin with one-dimensional indexers.

Creating One-Dimensional Indexers

A one-dimensional indexer has this general form:

```
element-type this[int index] {
  // The get accessor
  get {
     // return the value specified by index
   }

  // The set accessor
  set {
    // set the value specified by index
  }
}
```

Here, *element-type* is the element type of the indexer. Thus, each element accessed by the indexer will be of type *element-type.* This type corresponds to the element type of an array. The parameter *index* receives the index of the element being accessed. Technically, this parameter does not have to be of type **int**, but since indexers are typically used to provide array indexing, an integer type is customary.

Inside the body of the indexer two *accessors* are defined that are called **get** and **set**. An accessor is similar to a method except that it does not declare a return type or parameters. The accessors are automatically called when the indexer is used, and both accessors receive *index* as a parameter. If the indexer is on the left side of an assignment statement, then the **set** accessor is called and the element specified by *index* must be set. Otherwise, the **get** accessor is called and the value associated with *index* must be returned. The **set** method also receives an implicit parameter called **value**, which contains the value being assigned to the specified index.

One of the benefits of an indexer is that you can control precisely how an array is accessed, heading off improper access. Here is an example. In the following program, the **FailSoftArray** class implements an array that traps boundary errors, thus preventing runtime exceptions if the array is indexed out-of-bounds. This is accomplished by encapsulating the array as a private member of a class, allowing access to the array only through the indexer. With this approach, any attempt to access the array beyond its boundaries can be prevented, with such an attempt failing gracefully (resulting in a "soft landing" rather than a "crash"). Since **FailSoftArray** uses an indexer, the array can be accessed using the normal array notation.

```
// Use an indexer to create a fail-soft array.

using System;

class FailSoftArray {
  int[] a;    // reference to underlying array

  public int Length; // Length is public

  public bool ErrFlag; // indicates outcome of last operation

  // Construct array given its size.
  public FailSoftArray(int size) {
    a = new int[size];
    Length = size;
  }

  // This is the indexer for FailSoftArray.
  public int this[int index] {
    // This is the get accessor.
    get {
      if(ok(index)) {
        ErrFlag = false;
        return a[index];
      } else {
        ErrFlag = true;
        return 0;
      }
    }
```

```
    // This is the set accessor.
    set {
      if(ok(index)) {
        a[index] = value;
        ErrFlag = false;
      }
      else ErrFlag = true;
    }
  }

  // Return true if index is within bounds.
  private bool ok(int index) {
   if(index >= 0 & index < Length) return true;
   return false;
  }
}

// Demonstrate the fail-soft array.
class FSDemo {
  static void Main() {
    FailSoftArray fs = new FailSoftArray(5);
    int x;

    // Show quiet failures.
    Console.WriteLine("Fail quietly.");
    for(int i=0; i < (fs.Length * 2); i++)
      fs[i] = i*10;

    for(int i=0; i < (fs.Length * 2); i++) {
      x = fs[i];
      if(x != -1) Console.Write(x + " ");
    }
    Console.WriteLine();

    // Now, display failures.
    Console.WriteLine("\nFail with error reports.");
    for(int i=0; i < (fs.Length * 2); i++) {
      fs[i] = i*10;
      if(fs.ErrFlag)
        Console.WriteLine("fs[" + i + "] out-of-bounds");
    }

    for(int i=0; i < (fs.Length * 2); i++) {
      x = fs[i];
      if(!fs.ErrFlag) Console.Write(x + " ");
      else
        Console.WriteLine("fs[" + i + "] out-of-bounds");
    }
  }
}
```

The output from the program is shown here:

```
Fail quietly.
0 10 20 30 40 0 0 0 0 0
```

```
Fail with error reports.
fs[5] out-of-bounds
fs[6] out-of-bounds
fs[7] out-of-bounds
fs[8] out-of-bounds
fs[9] out-of-bounds
0 10 20 30 40 fs[5] out-of-bounds
fs[6] out-of-bounds
fs[7] out-of-bounds
fs[8] out-of-bounds
fs[9] out-of-bounds
```

The indexer prevents the array boundaries from being overrun. Let's look closely at each part of the indexer. It begins with this line:

```
public int this[int index] {
```

This declares an indexer that operates on **int** elements. The index is passed in *index*. The indexer is public, allowing it to be used by code outside of its class.

The **get** accessor is shown here:

```
get {
  if(ok(index)) {
    ErrFlag = false;
    return a[index];
  } else {
    ErrFlag = true;
    return 0;
  }
}
```

The **get** accessor prevents array boundary errors by first confirming that the index is not out-of-bounds. This range check is performed by the **ok()** method, which returns true if the index is valid and false otherwise. If the specified index is within bounds, the element corresponding to the index is returned. If it is out of bounds, no operation takes place and no overrun occurs. In this version of **FailSoftArray**, a variable called **ErrFlag** contains the outcome of each operation. This field can be examined after each operation to assess the success or failure of the operation. (In Chapter 13, you will see a better way to handle errors by using C#'s exception subsystem, but for now, using an error flag is an acceptable approach.)

The **set** accessor is shown here. It too prevents a boundary error.

```
set {
  if(ok(index)) {
    a[index] = value;
    ErrFlag = false;
  }
  else ErrFlag = true;
}
```

Here, if **index** is within bounds, the value passed in **value** is assigned to the corresponding element. Otherwise, **ErrFlag** is set to **true**. Recall that in an accessor method, **value** is an

implicit parameter that contains the value being assigned. You do not need to (nor can you) declare it.

It is not necessary for an indexer to support both **get** and **set**. You can create a read-only indexer by implementing only the **get** accessor. You can create a write-only indexer by implementing only **set**.

Indexers Can Be Overloaded

An indexer can be overloaded. The version executed will be the one that has the closest type-match between its parameter and the argument used as an index. Here is an example that overloads the **FailSoftArray** indexer for indexes of type **double**. The **double** indexer rounds its index to the nearest integer value.

```
// Overload the FailSoftArray indexer.

using System;

class FailSoftArray {
  int[] a;    // reference to underlying array

  public int Length; // Length is public

  public bool ErrFlag; // indicates outcome of last operation

  // Construct array given its size.
  public FailSoftArray(int size) {
    a = new int[size];
    Length = size;
  }

  // This is the int indexer for FailSoftArray.
  public int this[int index] {
    // This is the get accessor.
    get {
      if(ok(index)) {
        ErrFlag = false;
        return a[index];
      } else {
        ErrFlag = true;
        return 0;
      }
    }

    // This is the set accessor.
    set {
      if(ok(index)) {
        a[index] = value;
        ErrFlag = false;
      }
      else ErrFlag = true;
    }
  }
```

```
  /* This is another indexer for FailSoftArray.
     This index takes a double argument. It then
     rounds that argument to the nearest integer index. */
  public int this[double idx] {
    // This is the get accessor.
    get {
      int index;

      // Round to nearest int.
      if( (idx - (int) idx) < 0.5) index = (int) idx;
      else index = (int) idx + 1;

      if(ok(index)) {
        ErrFlag = false;
        return a[index];
      } else {
        ErrFlag = true;
        return 0;
      }
    }

    // This is the set accessor.
    set {
      int index;

      // Round to nearest int.
      if( (idx - (int) idx) < 0.5) index = (int) idx;
      else index = (int) idx + 1;

      if(ok(index)) {
        a[index] = value;
        ErrFlag = false;
      }
      else ErrFlag = true;
    }
  }

  // Return true if index is within bounds.
  private bool ok(int index) {
   if(index >= 0 & index < Length) return true;
   return false;
  }
}

// Demonstrate the fail-soft array.
class FSDemo {
  static void Main() {
    FailSoftArray fs = new FailSoftArray(5);

    // Put some values in fs.
    for(int i=0; i < fs.Length; i++)
      fs[i] = i;

    // Now index with ints and doubles.
    Console.WriteLine("fs[1]: " + fs[1]);
```

```
    Console.WriteLine("fs[2]: " + fs[2]);

    Console.WriteLine("fs[1.1]: " + fs[1.1]);
    Console.WriteLine("fs[1.6]: " + fs[1.6]);
  }
}
```

This program produces the following output:

```
fs[1]: 1
fs[2]: 2
fs[1.1]: 1
fs[1.6]: 2
```

As the output shows, the **double** indexes are rounded to their nearest integer value. Specifically, 1.1 is rounded to 1, and 1.6 is rounded to 2.

Although overloading an indexer as shown in this program is valid, it is not common. Most often, an indexer is overloaded to enable an object of a class to be used as an index, with the index computed in some special way.

Indexers Do Not Require an Underlying Array

It is important to understand that there is no requirement that an indexer actually operate on an array. It simply must provide functionality that appears "array-like" to the user of the indexer. For example, the following program has an indexer that acts like a read-only array that contains the powers of 2 from 0 to 15. Notice, however, that no actual array exists. Instead, the indexer simply computes the proper value for a given index.

```
// Indexers don't have to operate on actual arrays.

using System;

class PwrOfTwo {

  /* Access a logical array that contains
     the powers of 2 from 0 to 15. */
  public int this[int index] {
    // Compute and return power of 2.
    get {
      if((index >= 0) && (index < 16)) return pwr(index);
      else return -1;
    }

    // There is no set accessor.
  }

  int pwr(int p) {
    int result = 1;

    for(int i=0; i < p; i++)
      result *= 2;

    return result;
  }
}
```

```
class UsePwrOfTwo {
  static void Main() {
    PwrOfTwo pwr = new PwrOfTwo();

    Console.Write("First 8 powers of 2: ");
    for(int i=0; i < 8; i++)
      Console.Write(pwr[i] + " ");
    Console.WriteLine();

    Console.Write("Here are some errors: ");
    Console.Write(pwr[-1] + " " + pwr[17]);

    Console.WriteLine();
  }
}
```

The output from the program is shown here:

```
First 8 powers of 2: 1 2 4 8 16 32 64 128
Here are some errors: -1 -1
```

Notice that the indexer for **PwrOfTwo** includes a **get** accessor, but no **set** accessor. As explained, this means that the indexer is read-only. Thus, a **PwrOfTwo** object can be used on the right side of an assignment statement, but not on the left. For example, attempting to add this statement to the preceding program won't work:

```
pwr[0] = 11; // won't compile
```

This statement will cause a compilation error because no **set** accessor is defined for the indexer.

There are two important restrictions to using indexers. First, because an indexer does not define a storage location, a value produced by an indexer cannot be passed as a **ref** or **out** parameter to a method. Second, an indexer must be an instance member of its class; it cannot be declared **static**.

Multidimensional Indexers

You can create indexers for multidimensional arrays, too. For example, here is a two-dimensional fail-soft array. Pay close attention to the way that the indexer is declared.

```
// A two-dimensional fail-soft array.

using System;

class FailSoftArray2D {
  int[,] a; // reference to underlying 2D array
  int rows, cols; // dimensions
  public int Length; // Length is public

  public bool ErrFlag; // indicates outcome of last operation

  // Construct array given its dimensions.
  public FailSoftArray2D(int r, int c) {
```

```
    rows = r;
    cols = c;
    a = new int[rows, cols];
    Length = rows * cols;
  }

  // This is the indexer for FailSoftArray2D.
  public int this[int index1, int index2] {
    // This is the get accessor.
    get {
      if(ok(index1, index2)) {
        ErrFlag = false;
        return a[index1, index2];
      } else {
        ErrFlag = true;
        return 0;
      }
    }

    // This is the set accessor.
    set {
      if(ok(index1, index2)) {
        a[index1, index2] = value;
        ErrFlag = false;
      }
      else ErrFlag = true;
    }
  }

  // Return true if indexes are within bounds.
  private bool ok(int index1, int index2) {
   if(index1 >= 0 & index1 < rows &
      index2 >= 0 & index2 < cols)
         return true;

   return false;
  }
}

// Demonstrate a 2D indexer.
class TwoDIndexerDemo {
  static void Main() {
    FailSoftArray2D fs = new FailSoftArray2D(3, 5);
    int x;

    // Show quiet failures.
    Console.WriteLine("Fail quietly.");
    for(int i=0; i < 6; i++)
      fs[i, i] = i*10;

    for(int i=0; i < 6; i++) {
      x = fs[i,i];
      if(x != -1) Console.Write(x + " ");
    }
```

```
    Console.WriteLine();

    // Now, display failures.
    Console.WriteLine("\nFail with error reports.");
    for(int i=0; i < 6; i++) {
      fs[i,i] = i*10;
      if(fs.ErrFlag)
        Console.WriteLine("fs[" + i + ", " + i + "] out-of-bounds");
    }

    for(int i=0; i < 6; i++) {
      x = fs[i,i];
      if(!fs.ErrFlag) Console.Write(x + " ");
      else
        Console.WriteLine("fs[" + i + ", " + i + "] out-of-bounds");
    }
  }
}
```

The output from this program is shown here:

```
Fail quietly.
0 10 20 0 0 0

Fail with error reports.
fs[3, 3] out-of-bounds
fs[4, 4] out-of-bounds
fs[5, 5] out-of-bounds
0 10 20 fs[3, 3] out-of-bounds
fs[4, 4] out-of-bounds
fs[5, 5] out-of-bounds
```

Properties

Another type of class member is the *property.* As a general rule, a property combines a field with the methods that access it. As some examples earlier in this book have shown, you will often want to create a field that is available to users of an object, but you want to maintain control over the operations allowed on that field. For instance, you might want to limit the range of values that can be assigned to that field. While it is possible to accomplish this goal through the use of a private variable along with methods to access its value, a property offers a better, more streamlined approach.

Properties are similar to indexers. A property consists of a name along with **get** and **set** accessors. The accessors are used to get and set the value of a variable. The key benefit of a property is that its name can be used in expressions and assignments like a normal variable, but in actuality the **get** and **set** accessors are automatically invoked. This is similar to the way that an indexer's **get** and **set** accessors are automatically used.

The general form of a property is shown here:

```
type name {
  get {
     // get accessor code
  }
```

```
    set {
      // set accessor code
    }
  }
```

Here, *type* specifies the type of the property, such as **int**, and *name* is the name of the property. Once the property has been defined, any use of *name* results in a call to its appropriate accessor. The **set** accessor automatically receives a parameter called **value** that contains the value being assigned to the property.

It is important to understand that properties do not define storage locations. Instead, a property typically manages access to a field. It does not, itself, provide that field. The field must be specified independently of the property. (The exception is the *auto-implemented* property added by C# 3.0, which is described shortly.)

Here is a simple example that defines a property called **MyProp**, which is used to access the field **prop**. In this case, the property allows only positive values to be assigned.

```
// A simple property example.

using System;

class SimpProp {
  int prop; // field being managed by MyProp

  public SimpProp() { prop = 0; }

  /* This is the property that supports access to
     the private instance variable prop. It
     allows only positive values. */
  public int MyProp {
    get {
      return prop;
    }
    set {
      if(value >= 0) prop = value;
    }
  }
}

// Demonstrate a property.
class PropertyDemo {
  static void Main() {
    SimpProp ob = new SimpProp();

    Console.WriteLine("Original value of ob.MyProp: " + ob.MyProp);

    ob.MyProp = 100; // assign value
    Console.WriteLine("Value of ob.MyProp: " + ob.MyProp);

    // Can't assign negative value to prop.
    Console.WriteLine("Attempting to assign -10 to ob.MyProp");
    ob.MyProp = -10;
    Console.WriteLine("Value of ob.MyProp: " + ob.MyProp);
  }
}
```

Output from this program is shown here:

```
Original value of ob.MyProp: 0
Value of ob.MyProp: 100
Attempting to assign -10 to ob.MyProp
Value of ob.MyProp: 100
```

Let's examine this program carefully. The program defines one private field, called **prop**, and a property called **MyProp** that manages access to **prop**. As explained, a property by itself does not define a storage location. Instead, most properties simply manage access to a field. Furthermore, because **prop** is private, it can be accessed *only* through **MyProp**.

The property **MyProp** is specified as **public** so it can be accessed by code outside of its class. This makes sense because it provides access to **prop**, which is private. The **get** accessor simply returns the value of **prop**. The **set** accessor sets the value of **prop** if and only if that value is positive. Thus, the **MyProp** property controls what values **prop** can have. This is the essence of why properties are important.

The type of property defined by **MyProp** is called a read-write property because it allows its underlying field to be read and written. It is possible, however, to create read-only and write-only properties. To create a read-only property, define only a **get** accessor. To define a write-only property, define only a **set** accessor.

You can use a property to further improve the fail-soft array class. As you know, all arrays have a **Length** property associated with them. Up to now, the **FailSoftArray** class simply used a public integer field called **Length** for this purpose. This is not good practice, though, because it allows **Length** to be set to some value other than the length of the fail-soft array. (For example, a malicious programmer could intentionally corrupt its value.) We can remedy this situation by transforming **Length** into a read-only property, as shown in the following version of **FailSoftArray**:

```
// Add Length property to FailSoftArray.

using System;

class FailSoftArray {
  int[] a; // reference to underlying array
  int len; // length of array -- underlies Length property

  public bool ErrFlag; // indicates outcome of last operation

  // Construct array given its size.
  public FailSoftArray(int size) {
    a = new int[size];
    len = size;
  }

  // Read-only Length property.
  public int Length {
    get {
      return len;
    }
  }
```

```
  // This is the indexer for FailSoftArray.
  public int this[int index] {
    // This is the get accessor.
    get {
      if(ok(index)) {
        ErrFlag = false;
        return a[index];
      } else {
        ErrFlag = true;
        return 0;
      }
    }

    // This is the set accessor.
    set {
      if(ok(index)) {
        a[index] = value;
        ErrFlag = false;
      }
      else ErrFlag = true;
    }
  }

  // Return true if index is within bounds.
  private bool ok(int index) {
   if(index >= 0 & index < Length) return true;
   return false;
  }
}

// Demonstrate the improved fail-soft array.
class ImprovedFSDemo {
  static void Main() {
    FailSoftArray fs = new FailSoftArray(5);
    int x;

    // Can read Length.
    for(int i=0; i < fs.Length; i++)
      fs[i] = i*10;

    for(int i=0; i < fs.Length; i++) {
      x = fs[i];
      if(x != -1) Console.Write(x + " ");
    }
    Console.WriteLine();

    // fs.Length = 10; // Error, illegal!
  }
}
```

Length is now a property that uses the private variable **len** for its storage. **Length** defines only a **get** accessor, which means that it is read-only. Thus, **Length** can be read, but not

changed. To prove this to yourself, try removing the comment symbol preceding this line in the program:

```
// fs.Length = 10; // Error, illegal!
```

When you try to compile, you will receive an error message stating that **Length** is read-only.

Although the addition of the **Length** property improves **FailSoftArray**, it is not the only improvement that properties can make. The **ErrFlag** member is also a prime candidate for conversion into a property since access to it should also be limited to read-only. Here is the final improvement of **FailSafeArray**. It creates a property called **Error** that uses the original **ErrFlag** variable as its storage, and **ErrFlag** is made private to **FailSoftArray**.

```
// Convert ErrFlag into a property.

using System;

class FailSoftArray {
  int[] a; // reference to underlying array
  int len; // length of array

  bool ErrFlag; // now private

  // Construct array given its size.
  public FailSoftArray(int size) {
    a = new int[size];
    len = size;
  }

  // Read-only Length property.
  public int Length {
    get {
      return len;
    }
  }

  // Read-only Error property.
  public bool Error {
    get {
      return ErrFlag;
    }
  }

  // This is the indexer for FailSoftArray.
  public int this[int index] {
    // This is the get accessor.
    get {
      if(ok(index)) {
        ErrFlag = false;
        return a[index];
      } else {
        ErrFlag = true;
        return 0;
      }
    }
```

```
    // This is the set accessor.
    set {
      if(ok(index)) {
        a[index] = value;
        ErrFlag = false;
      }
      else ErrFlag = true;
    }
  }

  // Return true if index is within bounds.
  private bool ok(int index) {
   if(index >= 0 & index < Length) return true;
   return false;
  }
}

// Demonstrate the improved fail-soft array.
class FinalFSDemo {
  static void Main() {
    FailSoftArray fs = new FailSoftArray(5);

    // Use Error property.
    for(int i=0; i < fs.Length + 1; i++) {
      fs[i] = i*10;
      if(fs.Error)
        Console.WriteLine("Error with index " + i);
    }
  }
}
```

The creation of the **Error** property has caused two changes to be made to **FailSoftArray**. First, **ErrFlag** has been made private because it is now used as the underlying storage for the **Error** property. Thus, it won't be available directly. Second, the read-only **Error** property has been added. Now, programs that need to detect errors will interrogate **Error**. This is demonstrated in **Main( )**, where a boundary error is intentionally generated, and the **Error** property is used to detect it.

Auto-Implemented Properties

Beginning with C# 3.0, it is possible to implement very simple properties without having to explicitly define the variable managed by the property. Instead, you can let the compiler automatically supply the underlying variable. This is called an *auto-implemented property.* It has the following general form:

type name { get; set; }

Here, *type* specifies the type of the property and *name* specifies the name. Notice that **get** and **set** are immediately followed by a semicolon. The accessors for an auto-implemented property have no bodies. This syntax tells the compiler to automatically create a storage location (sometimes referred to as a *backing field*) that holds the value. This variable is not named and is not directly available to you. Instead, it can be accessed only through the property.

Here is how a property called **UserCount** is declared using an auto-implemented property:

```
public int UserCount { get; set; }
```

Notice that no variable is explicitly declared. As explained, the compiler automatically generates an anonymous field that holds the value. Otherwise, **UserCount** acts like and is used like any other property.

Unlike normal properties, an auto-implemented property cannot be read-only or write-only. Both the **get** and **set** must be specified in all cases. However, you can approximate the same effect by declaring either **get** or **set** as **private**, as explained in "Use Access Modifiers with Accessors" later in this chapter.

Although auto-implemented properties offer convenience, their use is limited to those cases in which you do not need control over the getting or setting of the backing field. Remember, you cannot access the backing field directly. This means that there is no way to constrain the value an auto-implemented property can have. Thus, auto-implemented properties simply let the name of the property act as a proxy for the field, itself. However, sometimes this is exactly what you want. Also, they can be very useful in cases in which properties are used to expose functionality to a third party, possibly through a design tool.

Use Object Initializers with Properties

As discussed in Chapter 8, C# 3.0 adds a new feature called an *object initializer*, which provides an alternative to explicitly calling a constructor when creating an object. When using object initializers, you specify initial values for the fields and/or properties that you want to initialize. Furthermore, the object initializer syntax is the same for both properties or fields. For example, here is the object initializer demonstration program from Chapter 8, reworked to show the use of object initializers with properties. Recall that the version shown in Chapter 8 used fields. The only difference between this version of the program and the one shown in Chapter 8 is that **Count** and **Str** have been converted from fields into properties. The object initializer syntax is unchanged.

```
// Use object initializers with properties.

using System;

class MyClass {
  // These are now properties.
  public int Count { get; set; }
  public string Str { get; set; }
}

class ObjInitDemo {
  static void Main() {
    // Construct a MyClass object by using object initializers.
    MyClass obj = new MyClass { Count = 100, Str = "Testing" };

    Console.WriteLine(obj.Count + " " + obj.Str);
  }
}
```

As you can see, the properties **Count** and **Str** are set via object initializer expressions. The output is the same as that produced by the program in Chapter 8 and is shown here:

```
100 Testing
```

As explained in Chapter 8, the object initializer syntax is most useful when working with anonymous types generated by a LINQ expression. In most other cases, you will use the normal constructor syntax.

Property Restrictions

Properties have some important restrictions. First, because a property does not define a storage location, it cannot be passed as a **ref** or **out** parameter to a method. Second, you cannot overload a property. (You *can* have two different properties that both access the same variable, but this would be unusual.) Finally, a property should not alter the state of the underlying variable when the **get** accessor is called. Although this rule is not enforced by the compiler, violating it is semantically wrong. A **get** operation should be nonintrusive.

Use Access Modifiers with Accessors

By default, the **set** and **get** accessors have the same accessibility as the indexer or property of which they are a part. For example, if the property is declared **public**, then by default the **get** and **set** accessors are also public. It is possible, however, to give **set** or **get** its own access modifier, such as **private**. In all cases, the access modifier for an accessor must be more restrictive then the access specification of its property or indexer.

There are a number of reasons why you may want to restrict the accessibility of an accessor. For example, you might want to let anyone obtain the value of a property, but allow only members of its class to set the property. To do this, declare the **set** accessor as **private**. For example, here is a property called **MyProp** that has its **set** accessor specified as **private**.

```
// Use an access modifier with an accessor.

using System;

class PropAccess {
  int prop; // field being managed by MyProp

  public PropAccess() { prop = 0; }

  /* This is the property that supports access to
     the private instance variable prop. It allows
     any code to obtain the value of prop, but only
     other class members can set the value of prop. */
  public int MyProp {
    get {
      return prop;
    }
    private set { // now, private
      prop = value;
    }
  }
```

```
  // This class member increments the value of MyProp.
  public void IncrProp() {
    MyProp++; // OK, in same class.
  }
}

// Demonstrate accessor access modifier.
class PropAccessDemo {
  static void Main() {
    PropAccess ob = new PropAccess();

    Console.WriteLine("Original value of ob.MyProp: " + ob.MyProp);

//    ob.MyProp = 100; // can't access set

    ob.IncrProp();
    Console.WriteLine("Value of ob.MyProp after increment: "
                      + ob.MyProp);
  }
}
```

In the **PropAccess** class, the **set** accessor is specified **private**. This means that it can be accessed by other class members, such as **IncrProp()**, but it cannot be accessed by code outside of **PropAccess**. This is why the attempt to assign **ob.MyProp** a value inside **PropAccessDemo** is commented out.

Perhaps the most important use of restricting an accessor's access is found when working with auto-implemented properties. As explained, it is not possible to create a read-only or write-only auto-implemented property because both the **get** and **set** accessors must be specified when the auto-implemented property is declared. However, you can gain much the same effect by declaring either **get** or **set** as **private**. For example, this declares what is effectively a read-only, auto-implemented **Length** property for the **FailSoftArray** class shown earlier.

```
public int Length { get; private set; }
```

Because **set** is **private**, **Length** can be set only by code within its class. Outside its class, an attempt to change **Length** is illegal. Thus, outside its class, **Length** is effectively read-only. The same technique can also be applied to the **Error** property, like this:

```
public bool Error { get; private set; }
```

This allows **Error** to be read, but not set, by code outside **FailSoftArray**.

To try the auto-implemented version of **Length** and **Error** with **FailSoftArray**, first remove the **len** and **ErrFlag** variables. They are no longer needed. Then, replace each use of **len** inside **FailSoftArray** with **Length** and each use of **ErrFlag** with **Error**. Here is the updated version of **FailSoftArray** along with a **Main()** method to demonstrate it:

```
// Use read-only, auto-implemented properties for Length and Error.

using System;
```

```
class FailSoftArray {
  int[] a; // reference to underlying array

  // Construct array given its size.
  public FailSoftArray(int size) {
    a = new int[size];
    Length = size;
  }

  // An auto-implemented, read-only Length property.
  public int Length { get; private set; }

  // An auto-implemented, read-only Error property.
  public bool Error { get; private set; }

  // This is the indexer for FailSoftArray.
  public int this[int index] {
    // This is the get accessor.
    get {
      if(ok(index)) {
        Error = false;
        return a[index];
      } else {
        Error = true;
        return 0;
      }
    }

    // This is the set accessor.
    set {
      if(ok(index)) {
        a[index] = value;
        Error = false;
      }
      else Error = true;
    }
  }

  // Return true if index is within bounds.
  private bool ok(int index) {
   if(index >= 0 & index < Length) return true;
   return false;
  }
}

// Demonstrate the improved fail-soft array.
class FinalFSDemo {
  static void Main() {
    FailSoftArray fs = new FailSoftArray(5);

    // Use Error property.
    for(int i=0; i < fs.Length + 1; i++) {
      fs[i] = i*10;
```

```
      if(fs.Error)
        Console.WriteLine("Error with index " + i);
    }
  }
}
```

This version of **FailSoftArray** works the same as the previous version, but it does not contain the explicitly declared backing fields.

Here are some restrictions that apply to using access modifiers with accessors. First, only the **set** or **get** accessor can be modified, not both. Furthermore, the access modifier must be more restrictive than the access level of the property or indexer. Finally, an access modifier cannot be used when declaring an accessor within an interface or when implementing an accessor specified by an interface. (Interfaces are described in Chapter 12.)

Using Indexers and Properties

Although the preceding examples have demonstrated the basic mechanism of indexers and properties, they haven't displayed their full power. To conclude this chapter, a class called **RangeArray** is developed that uses indexers and properties to create an array type in which the index range of the array is determined by the programmer.

As you know, in C# all arrays begin indexing at zero. However, some applications would benefit from an array that allows indexes to begin at any arbitrary point. For example, in some situations it might be more convenient for an array to begin indexing with 1. In another situation, it might be beneficial to allow negative indexes, such as an array that runs from –5 to 5. The **RangeArray** class developed here allows these and other types of indexing.

Using **RangeArray**, you can write code like this:

```
RangeArray ra = new RangeArray(-5, 10); // array with indexes from -5 to 10

for(int i=-5; i <= 10; i++) ra[i] = i; // index from -5 to 10
```

As you can guess, the first line constructs a **RangeArray** that runs from –5 to 10, inclusive. The first argument specifies the beginning index. The second argument specifies the ending index. Once **ra** has been constructed, it can be indexed from –5 to 10.

The entire **RangeArray** class is shown here, along with **RangeArrayDemo**, which demonstrates the array. As implemented here, **RangeArray** supports arrays of **int**, but you can change the data type, if desired.

```
/* Create a specifiable range array class.
   The RangeArray class allows indexing to begin at
   some value other than 0. When you create a RangeArray,
   you specify the beginning and ending index. Negative
   indexes are also allowed. For example, you can create
   arrays that index from -5 to 5, 1 to 10, or 50 to 56.
*/

using System;

class RangeArray {
  // Private data.
  int[] a; // reference to underlying array
```

```
  int lowerBound; // smallest index
  int upperBound; // largest index

  // An auto-implemented, read-only Length property.
  public int Length { get; private set; }

  // An auto-implemented, read-only Error property.
  public bool Error { get; private set; }

  // Construct array given its size.
  public RangeArray(int low, int high) {
    high++;
    if(high <= low) {
      Console.WriteLine("Invalid Indices");
      high = 1; // create a minimal array for safety
      low = 0;
    }
    a = new int[high - low];
    Length = high - low;

    lowerBound = low;
    upperBound = --high;
  }

  // This is the indexer for RangeArray.
  public int this[int index] {
    // This is the get accessor.
    get {
      if(ok(index)) {
        Error = false;
        return a[index - lowerBound];
      } else {
        Error = true;
        return 0;
      }
    }

    // This is the set accessor.
    set {
      if(ok(index)) {
        a[index - lowerBound] = value;
        Error = false;
      }
      else Error = true;
    }
  }

  // Return true if index is within bounds.
  private bool ok(int index) {
    if(index >= lowerBound & index <= upperBound) return true;
    return false;
  }
}
```

```
// Demonstrate the index-range array.
class RangeArrayDemo {
  static void Main() {
    RangeArray ra = new RangeArray(-5, 5);
    RangeArray ra2 = new RangeArray(1, 10);
    RangeArray ra3 = new RangeArray(-20, -12);

    // Demonstrate ra.
    Console.WriteLine("Length of ra: " + ra.Length);

    for(int i = -5; i <= 5; i++)
      ra[i] = i;

    Console.Write("Contents of ra: ");
    for(int i = -5; i <= 5; i++)
      Console.Write(ra[i] + " ");

    Console.WriteLine("\n");

    // Demonstrate ra2.
    Console.WriteLine("Length of ra2: " + ra2.Length);

    for(int i = 1; i <= 10; i++)
      ra2[i] = i;

    Console.Write("Contents of ra2: ");
    for(int i = 1; i <= 10; i++)
      Console.Write(ra2[i] + " ");

    Console.WriteLine("\n");

    // Demonstrate ra3.
    Console.WriteLine("Length of ra3: " + ra3.Length);

    for(int i = -20; i <= -12; i++)
      ra3[i] = i;

    Console.Write("Contents of ra3: ");
    for(int i = -20; i <= -12; i++)
      Console.Write(ra3[i] + " ");

    Console.WriteLine("\n");
  }
}
```

The output from the program is shown here:

```
Length of ra: 11
Contents of ra: -5 -4 -3 -2 -1 0 1 2 3 4 5

Length of ra2: 10
Contents of ra2: 1 2 3 4 5 6 7 8 9 10

Length of ra3: 9
Contents of ra3: -20 -19 -18 -17 -16 -15 -14 -13 -12
```

As the output verifies, objects of type **RangeArray** can be indexed in ways other than starting at zero. Let's look more closely at how **RangeArray** is implemented.

RangeArray begins by defining the following private instance variables:

```
// Private data.
int[] a; // reference to underlying array
int lowerBound; // smallest index
int upperBound; // largest index
```

The underlying array is referred to by **a**. This array is allocated by the **RangeArray** constructor. The index of the lower bound of the array is stored in **lowerBound**, and the index of the upper bound is stored in **upperBound**.

Next, the auto-implemented, read-only properties **Length** and **Error** are declared:

```
// An auto-implemented, read-only Length property.
public int Length { get; private set; }

// An auto-implemented, read-only Error property.
public bool Error { get; private set; }
```

Notice that for both properties, the **set** accessor is private. As explained earlier in this chapter, this results in what is effectively a read-only, auto-implemented property.

The **RangeArray** constructor is shown here:

```
// Construct array given its size.
public RangeArray(int low, int high) {
  high++;
  if(high <= low) {
    Console.WriteLine("Invalid Indices");
    high = 1; // create a minimal array for safety
    low = 0;
  }
  a = new int[high - low];
  Length = high - low;

  lowerBound = low;
  upperBound = --high;
}
```

A **RangeArray** is constructed by passing the lower bound index in **low** and the upper bound index in **high**. The value of **high** is then incremented because the indexes specified are inclusive. Next, a check is made to ensure that the upper index is greater than the lower index. If not, an error is reported and a one-element array is created. Next, storage for the array is allocated and assigned to **a**. Then the **Length** property is set equal to the number of elements in the array. Finally, **lowerBound** and **upperBound** are set.

Next, **RangeArray** implements its indexer, as shown here:

```
// This is the indexer for RangeArray.
public int this[int index] {
  // This is the get accessor.
  get {
    if(ok(index)) {
```

```
      Error = false;
      return a[index - lowerBound];
    } else {
      Error = true;
      return 0;
    }
  }

  // This is the set accessor.
  set {
    if(ok(index)) {
      a[index - lowerBound] = value;
      Error = false;
    }
    else Error = true;
  }
}
```

This indexer is similar to the one used by **FailSoftArray**, with one important exception. Notice the expression that indexes **a**. It is

```
index - lowerBound
```

This expression transforms the index passed in **index** into a zero-based index suitable for use on **a**. This expression works whether **lowerBound** is positive, negative, or zero.

The **ok( )** method is shown here:

```
// Return true if index is within bounds.
private bool ok(int index) {
  if(index >= lowerBound & index <= upperBound) return true;
  return false;
}
```

It is similar to the one used by **FailSoftArray** except that the range is checked by testing it against the values in **lowerBound** and **upperBound**.

RangeArray illustrates just one kind of custom array that you can create through the use of indexers and properties. There are, of course, several others. For example, you can create dynamic arrays, which expand and contract as needed, associative arrays, and sparse arrays. You might want to try creating one of these types of arrays as an exercise.

CHAPTER 11

Inheritance

Inheritance is one of the three foundational principles of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it.

In the language of C#, a class that is inherited is called a *base class.* The class that does the inheriting is called a *derived class.* Therefore, a derived class is a specialized version of a base class. It inherits all of the variables, methods, properties, and indexers defined by the base class and adds its own unique elements.

Inheritance Basics

C# supports inheritance by allowing one class to incorporate another class into its declaration. This is done by specifying a base class when a derived class is declared. Let's begin with an example. The following class called **TwoDShape** stores the width and height of a two-dimensional object, such as a square, rectangle, triangle, and so on.

```
// A class for two-dimensional objects.
class TwoDShape {
  public double Width;
  public double Height;

  public void ShowDim() {
    Console.WriteLine("Width and height are " +
                      Width + " and " + Height);
  }
}
```

TwoDShape can be used as a base class (that is, as a starting point) for classes that describe specific types of two-dimensional objects. For example, the following program uses **TwoDShape** to derive a class called **Triangle**. Pay close attention to the way that **Triangle** is declared.

```
// A simple class hierarchy.
```

```
using System;

// A class for two-dimensional objects.
class TwoDShape {
  public double Width;
  public double Height;

  public void ShowDim() {
    Console.WriteLine("Width and height are " +
                      Width + " and " + Height);
  }
}

// Triangle is derived from TwoDShape.
class Triangle : TwoDShape {
  public string Style; // style of triangle

  // Return area of triangle.
  public double Area() {
    return Width * Height / 2;
  }

  // Display a triangle's style.
  public void ShowStyle() {
    Console.WriteLine("Triangle is " + Style);
  }
}

class Shapes {
  static void Main() {
    Triangle t1 = new Triangle();
    Triangle t2 = new Triangle();

    t1.Width = 4.0;
    t1.Height = 4.0;
    t1.Style = "isosceles";

    t2.Width = 8.0;
    t2.Height = 12.0;
    t2.Style = "right";

    Console.WriteLine("Info for t1: ");
    t1.ShowStyle();
    t1.ShowDim();
    Console.WriteLine("Area is " + t1.Area());

    Console.WriteLine();

    Console.WriteLine("Info for t2: ");
    t2.ShowStyle();
    t2.ShowDim();
    Console.WriteLine("Area is " + t2.Area());
  }
}
```

The output from this program is shown here:

```
Info for t1:
Triangle is isosceles
Width and height are 4 and 4
Area is 8

Info for t2:
Triangle is right
Width and height are 8 and 12
Area is 48
```

The **Triangle** class creates a specific type of **TwoDShape**, in this case, a triangle. The **Triangle** class includes all of **TwoDShape** and adds the field **Style**, the method **Area( )**, and the method **ShowStyle( )**. A description of the type of triangle is stored in **Style**; **Area( )** computes and returns the area of the triangle; and **ShowStyle( )** displays the triangle style.

Notice the syntax that **Triangle** uses to inherit **TwoDShape**:

```
class Triangle : TwoDShape {
```

This syntax can be generalized. Whenever one class inherits another, the base class name follows the name of the derived class, separated by a colon. In C#, the syntax for inheriting a class is remarkably simple and easy to use.

Because **Triangle** includes all of the members of its base class, **TwoDShape**, it can access **Width** and **Height** inside **Area( )**. Also, inside **Main( )**, objects **t1** and **t2** can refer to **Width** and **Height** directly, as if they were part of **Triangle**. Figure 11-1 depicts conceptually how **TwoDShape** is incorporated into **Triangle**.

Even though **TwoDShape** is a base for **Triangle**, it is also a completely independent, stand-alone class. Being a base class for a derived class does not mean that the base class cannot be used by itself. For example, the following is perfectly valid:

```
TwoDShape shape = new TwoDShape();

shape.Width = 10;
shape.Height = 20;

shape.ShowDim();
```

Of course, an object of **TwoDShape** has no knowledge of or access to any classes derived from **TwoDShape**.

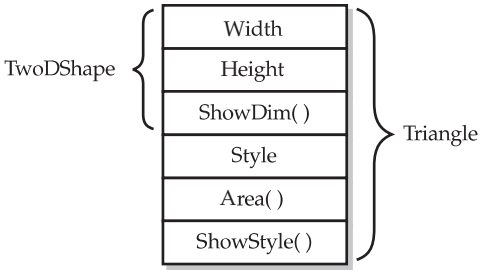


FIGURE 11-1 A conceptual depiction of the **Triangle** class

The general form of a **class** declaration that inherits a base class is shown here:

```
class derived-class-name : base-class-name {
  // body of class
}
```

You can specify only one base class for any derived class that you create. C# does not support the inheritance of multiple base classes into a single derived class. (This differs from C++, in which you can inherit multiple base classes. Be aware of this when converting C++ code to C#.) You can, however, create a hierarchy of inheritance in which a derived class becomes a base class of another derived class. (Of course, no class can be a base class of itself, either directly or indirectly.) In all cases, a derived class inherits all of the members of its base class. This includes instance variables, methods, properties, and indexers.

A major advantage of inheritance is that once you have created a base class that defines the attributes common to a set of objects, it can be used to create any number of more specific derived classes. Each derived class can precisely tailor its own classification. For example, here is another class derived from **TwoDShape** that encapsulates rectangles:

```
// A derived class of TwoDShape for rectangles.
class Rectangle : TwoDShape {
  // Return true if the rectangle is square.
  public bool IsSquare() {
    if(Width == Height) return true;
    return false;
  }

  // Return area of the rectangle.
  public double Area() {
    return Width * Height;
  }
}
```

The **Rectangle** class includes **TwoDShape** and adds the methods **IsSquare( )**, which determines if the rectangle is square, and **Area( )**, which computes the area of a rectangle.

Member Access and Inheritance

As explained in Chapter 8, members of a class are often declared private to prevent their unauthorized use or tampering. Inheriting a class *does not* overrule the private access restriction. Thus, even though a derived class includes all of the members of its base class, it cannot access those members of the base class that are private. For example, if, as shown here, **Width** and **Height** are made private in **TwoDShape**, then **Triangle** will not be able to access them:

```
// Access to private members is not inherited.

// This example will not compile.
using System;

// A class for two-dimensional objects.
class TwoDShape {
```

```
  double Width;  // now private
  double Height; // now private

  public void ShowDim() {
    Console.WriteLine("Width and height are " +
                      Width + " and " + Height);
  }
}

// Triangle is derived from TwoDShape.
class Triangle : TwoDShape {
  public string Style; // style of triangle

  // Return area of triangle.
  public double Area() {
    return Width * Height / 2; // Error, can't access private member
  }

  // Display a triangle's style.
  public void ShowStyle() {
    Console.WriteLine("Triangle is " + Style);
  }
}
```

The **Triangle** class will not compile because the use of **Width** and **Height** inside the **Area( )** method is illegal. Since **Width** and **Height** are now private, they are accessible only to other members of their own class. Derived classes have no access to them.

REMEMBER *A private class member will remain private to its class. It is not accessible to any code outside its class, including derived classes.*

At first, you might think that it is a serious restriction that derived classes do not have access to the private members of base classes because it would prevent the use of private members in many situations. However, this is not true; C# provides various solutions. One is to use **protected** members, which is described in the next section. A second is to use public properties to provide access to private data.

As explained in the previous chapter, a property allows you to manage access to an instance variable. For example, you can enforce constraints on its values, or you can make the variable read-only. By making a property public, but declaring its underlying variable private, a derived class can still use the property, but it cannot directly access the underlying private variable.

Here is a rewrite of the **TwoDShape** class that makes **Width** and **Height** into properties. In the process, it ensures that the values of **Width** and **Height** will be positive. This would allow you, for example, to specify the **Width** and **Height** using the coordinates of the shape in any quadrant of the Cartesian plane without having to first obtain their absolute values.

```
// Use public properties to set and get private members.

using System;

// A class for two-dimensional objects.
```

```
class TwoDShape {
  double pri_width;  // now private
  double pri_height; // now private

  // Properties for width and height.
  public double Width {
     get { return pri_width; }
     set { pri_width = value < 0 ? -value : value; }
  }

  public double Height {
     get { return pri_height; }
     set { pri_height = value < 0 ? -value : value; }
  }

  public void ShowDim() {
    Console.WriteLine("Width and height are " +
                      Width + " and " + Height);
  }
}

// A derived class of TwoDShape for triangles.
class Triangle : TwoDShape {
  public string Style; // style of triangle

  // Return area of triangle.
  public double Area() {
    return Width * Height / 2;
  }

  // Display a triangle's style.
  public void ShowStyle() {
    Console.WriteLine("Triangle is " + Style);
  }
}

class Shapes2 {
  static void Main() {
    Triangle t1 = new Triangle();
    Triangle t2 = new Triangle();

    t1.Width = 4.0;
    t1.Height = 4.0;
    t1.Style = "isosceles";

    t2.Width = 8.0;
    t2.Height = 12.0;
    t2.Style = "right";

    Console.WriteLine("Info for t1: ");
    t1.ShowStyle();
    t1.ShowDim();
    Console.WriteLine("Area is " + t1.Area());
```

```
    Console.WriteLine();

    Console.WriteLine("Info for t2: ");
    t2.ShowStyle();
    t2.ShowDim();
    Console.WriteLine("Area is " + t2.Area());
  }
}
```

In this version, the properties **Width** and **Height** provide access to the private members, **pri_width** and **pri_height**, which actually store the values. Therefore, even though **pri_width** and **pri_height** are private to **TwoDShape**, their values can still be set and obtained through their corresponding public properties.

When referring to base and derived classes, sometimes the terms *superclass* and *subclass* are used. These terms come from Java programming. What Java calls a superclass, C# calls a base class. What Java calls a subclass, C# calls a derived class. You will commonly hear both sets of terms applied to a class of either language, but this book will continue to use the standard C# terms. C++ also uses the base-class/derived-class terminology.

Using Protected Access

As just explained, a private member of a base class is not accessible to a derived class. This would seem to imply that if you wanted a derived class to have access to some member in the base class, it would need to be public. Of course, making the member public also makes it available to all other code, which may not be desirable. Fortunately, this implication is untrue because C# allows you to create a *protected member.* A protected member is public within a class hierarchy, but private outside that hierarchy.

A protected member is created by using the **protected** access modifier. When a member of a class is declared as **protected**, that member is, with one important exception, private. The exception occurs when a protected member is inherited. In this case, a protected member of the base class becomes a protected member of the derived class and is, therefore, accessible to the derived class. Therefore, by using **protected**, you can create class members that are private to their class but that can still be inherited and accessed by a derived class.

Here is a simple example that uses **protected**:

```
// Demonstrate protected.

using System;

class B {
  protected int i, j; // private to B, but accessible by D

  public void Set(int a, int b) {
    i = a;
    j = b;
  }

  public void Show() {
    Console.WriteLine(i + " " + j);
 }
}
```

```
class D : B {
  int k; // private

  // D can access B's i and j
  public void Setk() {
     k = i * j;
  }

  public void Showk() {
    Console.WriteLine(k);
  }
}

class ProtectedDemo {
  static void Main() {
    D ob = new D();

    ob.Set(2, 3); // OK, known to D
    ob.Show();    // OK, known to D

    ob.Setk();  // OK, part of D
    ob.Showk(); // OK, part of D
  }
}
```

In this example, because **B** is inherited by **D** and because **i** and **j** are declared as **protected** in **B**, the **Setk( )** method can access them. If **i** and **j** had been declared as private by **B**, then **D** would not have access to them, and the program would not compile.

Like **public** and **private**, **protected** status stays with a member no matter how many layers of inheritance are involved. Therefore, when a derived class is used as a base class for another derived class, any protected member of the initial base class that is inherited by the first derived class is also inherited as protected by a second derived class.

Although **protected** access is quite useful, it doesn't apply in all situations. For example, in the case of **TwoDShape** shown in the preceding section, we specifically want the **Width** and **Height** values to be publicly accessible. It's just that we want to manage the values they are assigned. Therefore, declaring them **protected** is not an option. In this case, the use of properties supplies the proper solution by controlling, rather than preventing, access. Remember, use **protected** when you want to create a member that is accessible throughout a class hierarchy, but otherwise private. To manage access to a value, use a property.

Constructors and Inheritance

In a hierarchy, it is possible for both base classes and derived classes to have their own constructors. This raises an important question: What constructor is responsible for building an object of the derived class? The one in the base class, the one in the derived class, or both? Here is the answer: The constructor for the base class constructs the base class portion of the object, and the constructor for the derived class constructs the derived class part. This makes sense because the base class has no knowledge of or access to any element in a derived class. Thus, their construction must be separate. The preceding examples have relied upon the default constructors created automatically by C#, so this was not an issue. However, in practice, most classes will define constructors. Here you will see how to handle this situation.

When only the derived class defines a constructor, the process is straightforward: Simply construct the derived class object. The base class portion of the object is constructed automatically using its default constructor. For example, here is a reworked version of **Triangle** that defines a constructor. It also makes **Style** private since it is now set by the constructor.

```
// Add a constructor to Triangle.

using System;

// A class for two-dimensional objects.
class TwoDShape {
  double pri_width;
  double pri_height;

  // Properties for Width and Height.
  public double Width {
     get { return pri_width; }
     set { pri_width = value < 0 ? -value : value; }
  }

  public double Height {
     get { return pri_height; }
     set { pri_height = value < 0 ? -value : value; }
  }

  public void ShowDim() {
    Console.WriteLine("Width and height are " +
                      Width + " and " + Height);
  }
}

// A derived class of TwoDShape for triangles.
class Triangle : TwoDShape {
  string Style;

  // Constructor.
  public Triangle(string s, double w, double h) {
    Width = w;  // init the base class
    Height = h; // init the base class

    Style = s;  // init the derived class
  }

  // Return area of triangle.
  public double Area() {
    return Width * Height / 2;
  }

  // Display a triangle's style.
  public void ShowStyle() {
    Console.WriteLine("Triangle is " + Style);
  }
}
```

```
class Shapes3 {
  static void Main() {
    Triangle t1 = new Triangle("isosceles", 4.0, 4.0);
    Triangle t2 = new Triangle("right", 8.0, 12.0);

    Console.WriteLine("Info for t1: ");
    t1.ShowStyle();
    t1.ShowDim();
    Console.WriteLine("Area is " + t1.Area());

    Console.WriteLine();

    Console.WriteLine("Info for t2: ");
    t2.ShowStyle();
    t2.ShowDim();
    Console.WriteLine("Area is " + t2.Area());
  }
}
```

Here, **Triangle**'s constructor initializes the members of **TwoDShape** that it inherits along with its own **Style** field.

When both the base class and the derived class define constructors, the process is a bit more complicated because both the base class and derived class constructors must be executed. In this case, you must use another of C#'s keywords, **base**, which has two uses. The first use is to call a base class constructor. The second is to access a member of the base class that has been hidden by a member of a derived class. Here, we will look at its first use.

Calling Base Class Constructors

A derived class can call a constructor defined in its base class by using an expanded form of the derived class' constructor declaration and the **base** keyword. The general form of this expanded declaration is shown here:

derived-constructor(*parameter-list*) : base(*arg-list*) {
  // body of constructor
}

Here, *arg-list* specifies any arguments needed by the constructor in the base class. Notice the placement of the colon.

To see how **base** is used, consider the version of **TwoDShape** in the following program. It defines a constructor that initializes the **Width** and **Height** properties. This constructor is then called by the **Triangle** constructor.

```
// Add constructor to TwoDShape.

using System;

// A class for two-dimensional objects.
class TwoDShape {
  double pri_width;
  double pri_height;
```

```
  // Constructor for TwoDShape.
  public TwoDShape(double w, double h) {
    Width = w;
    Height = h;
  }

  // Properties for Width and Height.
  public double Width {
     get { return pri_width; }
     set { pri_width = value < 0 ? -value : value; }
  }

  public double Height {
     get { return pri_height; }
     set { pri_height = value < 0 ? -value : value; }
  }

  public void ShowDim() {
    Console.WriteLine("Width and height are " +
                      Width + " and " + Height);
  }
}

 // A derived class of TwoDShape for triangles.
class Triangle : TwoDShape {
  string Style;

  // Call the base class constructor.
  public Triangle(string s, double w, double h) : base(w, h) {
    Style = s;
  }

  // Return area of triangle.
  public double Area() {
    return Width * Height / 2;
  }

  // Display a triangle's style.
  public void ShowStyle() {
    Console.WriteLine("Triangle is " + Style);
  }
}

class Shapes4 {
  static void Main() {
    Triangle t1 = new Triangle("isosceles", 4.0, 4.0);
    Triangle t2 = new Triangle("right", 8.0, 12.0);

    Console.WriteLine("Info for t1: ");
    t1.ShowStyle();
    t1.ShowDim();
    Console.WriteLine("Area is " + t1.Area());
```

```
    Console.WriteLine();

    Console.WriteLine("Info for t2: ");
    t2.ShowStyle();
    t2.ShowDim();
    Console.WriteLine("Area is " + t2.Area());
  }
}
```

Notice that the **Triangle** constructor is now declared as shown here.

```
public Triangle(string s, double w, double h) : base(w, h) {
```

In this version, **Triangle( )** calls **base** with the parameters **w** and **h**. This causes the **TwoDShape( )** constructor to be called, which initializes **Width** and **Height** using these values. **Triangle** no longer initializes these values itself. It need only initialize the value unique to it: **Style**. This leaves **TwoDShape** free to construct its subobject in any manner that it chooses. Furthermore, **TwoDShape** can add functionality about which existing derived classes have no knowledge, thus preventing existing code from breaking.

Any form of constructor defined by the base class can be called by **base**. The constructor executed will be the one that matches the arguments. For example, here are expanded versions of both **TwoDShape** and **Triangle** that include default constructors and constructors that take one argument.

```
// Add more constructors to TwoDShape.

using System;

class TwoDShape {
  double pri_width;
  double pri_height;

  // Default constructor.
  public TwoDShape() {
    Width = Height = 0.0;
  }

  // Constructor for TwoDShape.
  public TwoDShape(double w, double h) {
    Width = w;
    Height = h;
  }

  // Construct object with equal width and height.
  public TwoDShape(double x) {
    Width = Height = x;
  }

  // Properties for Width and Height.
  public double Width {
     get { return pri_width; }
     set { pri_width = value < 0 ? -value : value; }
  }
```

```
  public double Height {
     get { return pri_height; }
     set { pri_height = value < 0 ? -value : value; }
  }

  public void ShowDim() {
    Console.WriteLine("Width and height are " +
                      Width + " and " + Height);
  }
}

// A derived class of TwoDShape for triangles.
class Triangle : TwoDShape {
  string Style;

  /* A default constructor. This automatically invokes
     the default constructor of TwoDShape. */
  public Triangle() {
    Style = "null";
  }

  // Constructor that takes three arguments.
  public Triangle(string s, double w, double h) : base(w, h) {
    Style = s;
  }

  // Construct an isosceles triangle.
  public Triangle(double x) : base(x) {
    Style = "isosceles";
  }

  // Return area of triangle.
  public double Area() {
    return Width * Height / 2;
  }

  // Display a triangle's style.
  public void ShowStyle() {
    Console.WriteLine("Triangle is " + Style);
  }
}

class Shapes5 {
  static void Main() {
    Triangle t1 = new Triangle();
    Triangle t2 = new Triangle("right", 8.0, 12.0);
    Triangle t3 = new Triangle(4.0);

    t1 = t2;

    Console.WriteLine("Info for t1: ");
    t1.ShowStyle();
    t1.ShowDim();
    Console.WriteLine("Area is " + t1.Area());
```

```
    Console.WriteLine();

    Console.WriteLine("Info for t2: ");
    t2.ShowStyle();
    t2.ShowDim();
    Console.WriteLine("Area is " + t2.Area());

    Console.WriteLine();

    Console.WriteLine("Info for t3: ");
    t3.ShowStyle();
    t3.ShowDim();
    Console.WriteLine("Area is " + t3.Area());

    Console.WriteLine();
  }
}
```

Here is the output from this version:

```
Info for t1:
Triangle is right
Width and height are 8 and 12
Area is 48

Info for t2:
Triangle is right
Width and height are 8 and 12
Area is 48

Info for t3:
Triangle is isosceles
Width and height are 4 and 4
Area is 8
```

Let's review the key concepts behind **base**. When a derived class specifies a **base** clause, it is calling the constructor of its immediate base class. Thus, **base** always refers to the base class immediately above the calling class. This is true even in a multileveled hierarchy. You pass arguments to the base constructor by specifying them as arguments to **base**. If no **base** clause is present, then the base class' default constructor is called automatically.

Inheritance and Name Hiding

It is possible for a derived class to define a member that has the same name as a member in its base class. When this happens, the member in the base class is hidden within the derived class. While this is not technically an error in C#, the compiler will issue a warning message. This warning alerts you to the fact that a name is being hidden. If your intent is to hide a base class member, then to prevent this warning, the derived class member must be preceded by the **new** keyword. Understand that this use of **new** is separate and distinct from its use when creating an object instance.

Here is an example of name hiding:

```
// An example of inheritance-related name hiding.

using System;

class A {
  public int i = 0;
}

// Create a derived class.
class B : A {
  new int i; // this i hides the i in A

  public B(int b) {
    i = b; // i in B
  }

  public void Show() {
    Console.WriteLine("i in derived class: " + i);
  }
}

class NameHiding {
  static void Main() {
    B ob = new B(2);

    ob.Show();
  }
}
```

First, notice the use of **new** in this line.

```
new int i; // this i hides the i in A
```

In essence, it tells the compiler that you know a new variable called **i** is being created that hides the **i** in the base class **A**. If you leave **new** out, a warning is generated.

The output produced by this program is shown here:

```
i in derived class: 2
```

Since **B** defines its own instance variable called **i**, it hides the **i** in **A**. Therefore, when **Show( )** is invoked on an object of type **B**, the value of **i** as defined by **B** is displayed—not the one defined in **A**.

Using base to Access a Hidden Name

There is a second form of **base** that acts somewhat like **this**, except that it always refers to the base class of the derived class in which it is used. This usage has the following general form:

base.*member*

Here, *member* can be either a method or an instance variable. This form of **base** is most applicable to situations in which member names of a derived class hide members by the

same name in the base class. Consider this version of the class hierarchy from the preceding example:

```
// Using base to overcome name hiding.

using System;

class A {
  public int i = 0;
}

// Create a derived class.
class B : A {
  new int i; // this i hides the i in A

  public B(int a, int b) {
    base.i = a; // this uncovers the i in A
    i = b; // i in B
  }

  public void Show() {
    // This displays the i in A.
    Console.WriteLine("i in base class: " + base.i);

    // This displays the i in B.
    Console.WriteLine("i in derived class: " + i);
  }
}

class UncoverName {
  static void Main() {
    B ob = new B(1, 2);

    ob.Show();
  }
}
```

This program displays the following:

```
i in base class: 1
i in derived class: 2
```

Although the instance variable **i** in **B** hides the **i** in **A**, **base** allows access to the **i** defined in the base class.

Hidden methods can also be called through the use of **base**. For example, in the following code, class **B** inherits class **A**, and both **A** and **B** declare a method called **Show()**. Inside, **B**'s **Show()**, the version of **Show()** defined by **A** is called through the use of **base**.

```
// Call a hidden method.

using System;

class A {
```

```
  public int i = 0;

  // Show() in A
  public void Show() {
    Console.WriteLine("i in base class: " + i);
  }
}

// Create a derived class.
class B : A {
  new int i; // this i hides the i in A

  public B(int a, int b) {
    base.i = a; // this uncovers the i in A
    i = b; // i in B
  }

  // This hides Show() in A. Notice the use of new.
  new public void Show() {
    base.Show(); // this calls Show() in A

    // this displays the i in B
    Console.WriteLine("i in derived class: " + i);
  }
}

class UncoverName {
  static void Main() {
    B ob = new B(1, 2);

    ob.Show();
  }
}
```

The output from the program is shown here:

```
i in base class: 1
i in derived class: 2
```

As you can see, **base.Show( )** calls the base class version of **Show( )**.

One other point: Notice that **new** is used in this program to tell the compiler that you know a new method called **Show( )** is being declared that hides the **Show( )** in **A**.

Creating a Multilevel Hierarchy

Up to this point, we have been using simple class hierarchies consisting of only a base class and a derived class. However, you can build hierarchies that contain as many layers of inheritance as you like. As mentioned, it is perfectly acceptable to use a derived class as a base class of another. For example, given three classes called **A**, **B**, and **C**, **C** can be derived from **B**, which can be derived from **A**. When this type of situation occurs, each derived class inherits all of the traits found in all of its base classes. In this case, **C** inherits all aspects of **B** and **A**.

To see how a multilevel hierarchy can be useful, consider the following program. In it, the derived class **Triangle** is used as a base class to create the derived class called **ColorTriangle**. **ColorTriangle** inherits all of the traits of **Triangle** and **TwoDShape** and adds a field called **color**, which holds the color of the triangle.

```
// A multilevel hierarchy.

using System;

class TwoDShape {
  double pri_width;
  double pri_height;

  // Default constructor.
  public TwoDShape() {
    Width = Height = 0.0;
  }

  // Constructor for TwoDShape.
  public TwoDShape(double w, double h) {
    Width = w;
    Height = h;
  }

  // Construct object with equal width and height.
  public TwoDShape(double x) {
    Width = Height = x;
  }

  // Properties for Width and Height.
  public double Width {
     get { return pri_width; }
     set { pri_width = value < 0 ? -value : value; }
  }

  public double Height {
     get { return pri_height; }
     set { pri_height = value < 0 ? -value : value; }
  }

  public void ShowDim() {
    Console.WriteLine("Width and height are " +
                      Width + " and " + Height);
  }
}

// A derived class of TwoDShape for triangles.
class Triangle : TwoDShape {
  string Style; // private

  /* A default constructor. This invokes the default
     constructor of TwoDShape. */
  public Triangle() {
    Style = "null";
  }
```

```
  // Constructor.
  public Triangle(string s, double w, double h) : base(w, h) {
    Style = s;
  }

  // Construct an isosceles triangle.
  public Triangle(double x) : base(x) {
    Style = "isosceles";
  }

  // Return area of triangle.
  public double Area() {
    return Width * Height / 2;
  }

  // Display a triangle's style.
  public void ShowStyle() {
    Console.WriteLine("Triangle is " + Style);
  }
}

// Extend Triangle.
class ColorTriangle : Triangle {
  string color;

  public ColorTriangle(string c, string s,
                       double w, double h) : base(s, w, h) {
    color = c;
  }

  // Display the color.
  public void ShowColor() {
    Console.WriteLine("Color is " + color);
  }
}

class Shapes6 {
  static void Main() {
    ColorTriangle t1 =
         new ColorTriangle("Blue", "right", 8.0, 12.0);
    ColorTriangle t2 =
         new ColorTriangle("Red", "isosceles", 2.0, 2.0);

    Console.WriteLine("Info for t1: ");
    t1.ShowStyle();
    t1.ShowDim();
    t1.ShowColor();
    Console.WriteLine("Area is " + t1.Area());

    Console.WriteLine();

    Console.WriteLine("Info for t2: ");
    t2.ShowStyle();
    t2.ShowDim();
    t2.ShowColor();
```

```
    Console.WriteLine("Area is " + t2.Area());
  }
}
```

The output of this program is shown here:

```
Info for t1:
Triangle is right
Width and height are 8 and 12
Color is Blue
Area is 48

Info for t2:
Triangle is isosceles
Width and height are 2 and 2
Color is Red
Area is 2
```

Because of inheritance, **ColorTriangle** can make use of the previously defined classes of **Triangle** and **TwoDShape**, adding only the extra information it needs for its own, specific application. This is part of the value of inheritance; it allows the reuse of code.

This example illustrates one other important point: **base** always refers to the constructor in the closest base class. The **base** in **ColorTriangle** calls the constructor in **Triangle**. The **base** in **Triangle** calls the constructor in **TwoDShape**. In a class hierarchy, if a base class constructor requires parameters, then all derived classes must pass those parameters "up the line." This is true whether or not a derived class needs parameters of its own.

When Are Constructors Called?

In the foregoing discussion of inheritance and class hierarchies, an important question may have occurred to you: When a derived class object is created, whose constructor is executed first? The one in the derived class or the one defined by the base class? For example, given a derived class called **B** and a base class called **A**, is **A**'s constructor called before **B**'s, or vice versa? The answer is that in a class hierarchy, constructors are called in order of derivation, from base class to derived class. Furthermore, this order is the same whether or not **base** is used. If **base** is not used, then the default (parameterless) constructor of each base class will be executed. The following program illustrates the order of constructor execution:

```
// Demonstrate when constructors are called.

using System;

// Create a base class.
class A {
  public A() {
    Console.WriteLine("Constructing A.");
  }
}

// Create a class derived from A.
class B : A {
  public B() {
```

```
    Console.WriteLine("Constructing B.");
  }
}

// Create a class derived from B.
class C : B {
  public C() {
    Console.WriteLine("Constructing C.");
  }
}

class OrderOfConstruction {
  static void Main() {
    C c = new C();
  }
}
```

The output from this program is shown here:

```
Constructing A.
Constructing B.
Constructing C.
```

As you can see, the constructors are called in order of derivation.

If you think about it, it makes sense that constructors are executed in order of derivation. Because a base class has no knowledge of any derived class, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the derived class. Therefore, it must be executed first.

Base Class References and Derived Objects

As you know, C# is a strongly typed language. Aside from the standard conversions and automatic promotions that apply to its value types, type compatibility is strictly enforced. Therefore, a reference variable for one class type cannot normally refer to an object of another class type. For example, consider the following program that declares two classes that are identical in their composition:

```
// This program will not compile.

class X {
  int a;

  public X(int i) { a = i; }
}

class Y {
  int a;

  public Y(int i) { a = i; }
}

class IncompatibleRef {
  static void Main() {
```

```
    X x = new X(10);
    X x2;
    Y y = new Y(5);

    x2 = x; // OK, both of same type

    x2 = y; // Error, not of same type
  }
}
```

Here, even though class **X** and class **Y** are physically the same, it is not possible to assign a reference of type **Y** to a variable of type **X** because they have different types. Therefore, this line is incorrect because it causes a compile-time type mismatch:

```
x2 = y; // Error, not of same type
```

In general, an object reference variable can refer only to objects of its type.

There is, however, an important exception to C#'s strict type enforcement. A reference variable of a base class can be assigned a reference to an object of any class derived from that base class. This is legal because an instance of a derived type encapsulates an instance of the base type. Thus, a base class reference can refer to it. Here is an example:

```
// A base class reference can refer to a derived class object.

using System;

class X {
  public int a;

  public X(int i) {
    a = i;
  }
}

class Y : X {
  public int b;

  public Y(int i, int j) : base(j) {
    b = i;
  }
}

class BaseRef {
  static void Main() {
    X x = new X(10);
    X x2;
    Y y = new Y(5, 6);

    x2 = x; // OK, both of same type
    Console.WriteLine("x2.a: " + x2.a);

    x2 = y; // OK because Y is derived from X
    Console.WriteLine("x2.a: " + x2.a);
```

```
    // X references know only about X members
    x2.a = 19; // OK
//    x2.b = 27; // Error, X doesn't have a b member
  }
}
```

In this program, **Y** is derived from **X**. Now, the assignment

```
x2 = y; // OK because Y is derived from X
```

is permissible because a base class reference, **x2** in this case, can refer to a derived class object (which is the object referred to by **y**).

It is important to understand that it is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed. That is, when a reference to a derived class object is assigned to a base class reference variable, you will have access only to those parts of the object defined by the base class. This is why **x2** can't access **b** even when it refers to a **Y** object. This makes sense because the base class has no knowledge of what a derived class adds to it. This is why the last line of code in the program is commented out.

Although the preceding discussion may seem a bit esoteric, it has some important practical applications. One is described here. The other is discussed later in this chapter, when virtual methods are covered.

An important place where derived class references are assigned to base class variables is when constructors are called in a class hierarchy. As you know, it is common for a class to define a constructor that takes an object of its class as a parameter. This allows the class to construct a copy of an object. Classes derived from such a class can take advantage of this feature. For example, consider the following versions of **TwoDShape** and **Triangle**. Both add constructors that take an object as a parameter.

```
// Pass a derived class reference to a base class reference.

using System;

class TwoDShape {
  double pri_width;
  double pri_height;

  // Default constructor.
  public TwoDShape() {
    Width = Height = 0.0;
  }

  // Constructor for TwoDShape.
  public TwoDShape(double w, double h) {
    Width = w;
    Height = h;
  }

  // Construct object with equal width and height.
  public TwoDShape(double x) {
    Width = Height = x;
  }
```

```
  // Construct a copy of a TwoDShape object.
  public TwoDShape(TwoDShape ob) {
    Width = ob.Width;
    Height = ob.Height;
  }

  // Properties for Width and Height.
  public double Width {
     get { return pri_width; }
     set { pri_width = value < 0 ? -value : value; }
  }

  public double Height {
     get { return pri_height; }
     set { pri_height = value < 0 ? -value : value; }
  }

  public void ShowDim() {
    Console.WriteLine("Width and height are " +
                      Width + " and " + Height);
  }
}

// A derived class of TwoDShape for triangles.
class Triangle : TwoDShape {
  string Style;

  // A default constructor.
  public Triangle() {
    Style = "null";
  }

  // Constructor for Triangle.
  public Triangle(string s, double w, double h) : base(w, h) {
    Style = s;
  }

  // Construct an isosceles triangle.
  public Triangle(double x) : base(x) {
    Style = "isosceles";
  }

  // Construct a copy of a Triangle object.
  public Triangle(Triangle ob) : base(ob) {
    Style = ob.Style;
  }

  // Return area of triangle.
  public double Area() {
    return Width * Height / 2;
  }
```

```
  // Display a triangle's style.
  public void ShowStyle() {
    Console.WriteLine("Triangle is " + Style);
  }
}

class Shapes7 {
  static void Main() {
    Triangle t1 = new Triangle("right", 8.0, 12.0);

    // Make a copy of t1.
    Triangle t2 = new Triangle(t1);

    Console.WriteLine("Info for t1: ");
    t1.ShowStyle();
    t1.ShowDim();
    Console.WriteLine("Area is " + t1.Area());

    Console.WriteLine();

    Console.WriteLine("Info for t2: ");
    t2.ShowStyle();
    t2.ShowDim();
    Console.WriteLine("Area is " + t2.Area());
  }
}
```

In this program, **t2** is constructed from **t1** and is, thus, identical. The output is shown here:

```
Info for t1:
Triangle is right
Width and height are 8 and 12
Area is 48

Info for t2:
Triangle is right
Width and height are 8 and 12
Area is 48
```

Pay special attention to this **Triangle** constructor:

```
public Triangle(Triangle ob) : base(ob) {
  Style = ob.Style;
}
```

It receives an object of type **Triangle**, and it passes that object (through **base**) to this **TwoDShape** constructor:

```
public TwoDShape(TwoDShape ob) {
  Width = ob.Width;
  Height = ob.Height;
}
```

The key point is that **TwoDShape( )** is expecting a **TwoDShape** object. However, **Triangle( )** passes it a **Triangle** object. As explained, the reason this works is because a base class reference can refer to a derived class object. Thus, it is perfectly acceptable to pass **TwoDShape( )** a reference to an object of a class derived from **TwoDShape**. Because the **TwoDShape( )** constructor is initializing only those portions of the derived class object that are members of **TwoDShape**, it doesn't matter that the object might also contain other members added by derived classes.

Virtual Methods and Overriding

A *virtual method* is a method that is declared as **virtual** in a base class. The defining characteristic of a virtual method is that it can be redefined in one or more derived classes. Thus, each derived class can have its own version of a virtual method. Virtual methods are interesting because of what happens when one is called through a base class reference. In this situation, C# determines which version of the method to call based upon the *type* of the object *referred to* by the reference—and this determination is made *at runtime.* Thus, when different objects are referred to, different versions of the virtual method are executed. In other words, it is the type of the object being referred to (not the type of the reference) that determines which version of the virtual method will be executed. Therefore, if a base class contains a virtual method and classes are derived from that base class, then when different types of objects are referred to through a base class reference, different versions of the virtual method are executed.

You declare a method as virtual inside a base class by preceding its declaration with the keyword **virtual**. When a virtual method is redefined by a derived class, the **override** modifier is used. Thus, the process of redefining a virtual method inside a derived class is called *method overriding.* When overriding a method, the name, return type, and signature of the overriding method must be the same as the virtual method that is being overridden. Also, a virtual method cannot be specified as **static** or **abstract** (discussed later in this chapter).

Method overriding forms the basis for one of C#'s most powerful concepts: *dynamic method dispatch.* Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at runtime, rather than compile time. Dynamic method dispatch is important because this is how C# implements runtime polymorphism.

Here is an example that illustrates virtual methods and overriding:

```
// Demonstrate a virtual method.

using System;

class Base {
  // Create virtual method in the base class.
  public virtual void Who() {
    Console.WriteLine("Who() in Base");
  }
}

class Derived1 : Base {
  // Override Who() in a derived class.
  public override void Who() {
    Console.WriteLine("Who() in Derived1");
  }
}
```

```
class Derived2 : Base {
  // Override Who() again in another derived class.
  public override void Who() {
    Console.WriteLine("Who() in Derived2");
  }
}

class OverrideDemo {
  static void Main() {
    Base baseOb = new Base();
    Derived1 dOb1 = new Derived1();
    Derived2 dOb2 = new Derived2();

    Base baseRef; // a base class reference

    baseRef = baseOb;
    baseRef.Who();

    baseRef = dOb1;
    baseRef.Who();

    baseRef = dOb2;
    baseRef.Who();
  }
}
```

The output from the program is shown here:

```
Who() in Base
Who() in Derived1
Who() in Derived2
```

This program creates a base class called **Base** and two derived classes, called **Derived1** and **Derived2**. **Base** declares a method called **Who( )**, and the derived classes override it. Inside the **Main( )** method, objects of type **Base**, **Derived1**, and **Derived2** are declared. Also, a reference of type **Base**, called **baseRef**, is declared. The program then assigns a reference to each type of object to **baseRef** and uses that reference to call **Who( )**. As the output shows, the version of **Who( )** executed is determined by the type of object being referred to at the time of the call, not by the class type of **baseRef**.

It is not necessary to override a virtual method. If a derived class does not provide its own version of a virtual method, then the one in the base class is used. For example:

```
/* When a virtual method is not overridden,
   the base class method is used. */

using System;

class Base {
  // Create virtual method in the base class.
  public virtual void Who() {
    Console.WriteLine("Who() in Base");
  }
}
```

```
class Derived1 : Base {
  // Override Who() in a derived class.
  public override void Who() {
    Console.WriteLine("Who() in Derived1");
  }
}

class Derived2 : Base {
  // This class does not override Who().
}

class NoOverrideDemo {
  static void Main() {
    Base baseOb = new Base();
    Derived1 dOb1 = new Derived1();
    Derived2 dOb2 = new Derived2();

    Base baseRef; // a base class reference

    baseRef = baseOb;
    baseRef.Who();

    baseRef = dOb1;
    baseRef.Who();

    baseRef = dOb2;
    baseRef.Who(); // calls Base's Who()
  }
}
```

The output from this program is shown here:

```
Who() in Base
Who() in Derived1
Who() in Base
```

Here, **Derived2** does not override **Who( )**. Thus, when **Who( )** is called on a **Derived2** object, the **Who( )** in **Base** is executed.

In the case of a multilevel hierarchy, if a derived class does not override a virtual method, then, while moving up the hierarchy, the first override of the method that is encountered is the one executed. For example:

```
/*  In a multilevel hierarchy, the first override of a virtual
    method that is found while moving up the hierarchy is the
    one executed. */

using System;

class Base {
  // Create virtual method in the base class.
  public virtual void Who() {
    Console.WriteLine("Who() in Base");
```

```
  }
}

class Derived1 : Base {
  // Override Who() in a derived class.
  public override void Who() {
    Console.WriteLine("Who() in Derived1");
  }
}

class Derived2 : Derived1 {
  // This class also does not override Who().
}

class Derived3 : Derived2 {
  // This class does not override Who().
}

class NoOverrideDemo2 {
  static void Main() {
    Derived3 dOb = new Derived3();
    Base baseRef; // a base class reference

    baseRef = dOb;
    baseRef.Who(); // calls Derived1's Who()
  }
}
```

The output is shown here:

```
Who() in Derived1
```

Here, **Derived3** inherits **Derived2**, which inherits **Derived1**, which inherits **Base**. As the output verifies, since **Who( )** is not overridden by either **Derived3** or **Derived2**, it is the override of **Who( )** in **Derived1** that is executed, since it is the first version of **Who( )** that is found.

One other point: Properties can also be modified by the **virtual** keyword and overridden using **override**. The same is true for indexers.

Why Overridden Methods?

Overridden methods allow C# to support runtime polymorphism. Polymorphism is essential to object-oriented programming for one reason: It allows a general class to specify methods that will be common to all of its derivatives, while allowing derived classes to define the specific implementation of some or all of those methods. Overridden methods are another way that C# implements the "one interface, multiple methods" aspect of polymorphism.

Part of the key to applying polymorphism successfully is understanding that the base classes and derived classes form a hierarchy that moves from lesser to greater specialization. Used correctly, the base class provides all elements that a derived class can use directly. Through virtual methods, it also defines those methods that the derived class can implement on its own. This allows the derived class flexibility, yet still enforces a consistent interface. Thus, by combining inheritance with overridden methods, a base class can define the general form of the methods that will be used by all of its derived classes.

Applying Virtual Methods

To better understand the power of virtual methods, we will apply them to the **TwoDShape** class. In the preceding examples, each class derived from **TwoDShape** defines a method called **Area( )**. This suggests that it might be better to make **Area( )** a virtual method of the **TwoDShape** class, allowing each derived class to override it, defining how the area is calculated for the type of shape that the class encapsulates. The following program does this. For convenience, it also adds a name property to **TwoDShape**. (This makes it easier to demonstrate the classes.)

```
// Use virtual methods and polymorphism.

using System;

class TwoDShape {
  double pri_width;
  double pri_height;

  // A default constructor.
  public TwoDShape() {
    Width = Height = 0.0;
    name = "null";
  }

  // Parameterized constructor.
  public TwoDShape(double w, double h, string n) {
    Width = w;
    Height = h;
    name = n;
  }

  // Construct object with equal width and height.
  public TwoDShape(double x, string n) {
    Width = Height = x;
    name = n;
  }

  // Construct a copy of a TwoDShape object.
  public TwoDShape(TwoDShape ob) {
    Width = ob.Width;
    Height = ob.Height;
    name = ob.name;
  }

  // Properties for Width and Height.
  public double Width {
     get { return pri_width; }
     set { pri_width = value < 0 ? -value : value; }
  }

  public double Height {
     get { return pri_height; }
     set { pri_height = value < 0 ? -value : value; }
  }
```

```
  public string name { get; set; }

  public void ShowDim() {
    Console.WriteLine("Width and height are " +
                      Width + " and " + Height);
  }

  public virtual double Area() {
    Console.WriteLine("Area() must be overridden");
    return 0.0;
  }
}

// A derived class of TwoDShape for triangles.
class Triangle : TwoDShape {
  string Style;

  // A default constructor.
  public Triangle() {
    Style = "null";
  }

  // Constructor for Triangle.
  public Triangle(string s, double w, double h) :
    base(w, h, "triangle") {
      Style = s;
  }

  // Construct an isosceles triangle.
  public Triangle(double x) : base(x, "triangle") {
    Style = "isosceles";
  }

  // Construct a copy of a Triangle object.
  public Triangle(Triangle ob) : base(ob) {
    Style = ob.Style;
  }

  // Override Area() for Triangle.
  public override double Area() {
    return Width * Height / 2;
  }

  // Display a triangle's style.
  public void ShowStyle() {
    Console.WriteLine("Triangle is " + Style);
  }
}

// A derived class of TwoDShape for rectangles.
class Rectangle : TwoDShape {

  // Constructor for Rectangle.
  public Rectangle(double w, double h) :
    base(w, h, "rectangle"){ }
```

```
  // Construct a square.
  public Rectangle(double x) :
    base(x, "rectangle") { }

  // Construct a copy of a Rectangle object.
  public Rectangle(Rectangle ob) : base(ob) { }

  // Return true if the rectangle is square.
  public bool IsSquare() {
    if(Width == Height) return true;
    return false;
  }

  // Override Area() for Rectangle.
  public override double Area() {
    return Width * Height;
  }
}

class DynShapes {
  static void Main() {
    TwoDShape[] shapes = new TwoDShape[5];

    shapes[0] = new Triangle("right", 8.0, 12.0);
    shapes[1] = new Rectangle(10);
    shapes[2] = new Rectangle(10, 4);
    shapes[3] = new Triangle(7.0);
    shapes[4] = new TwoDShape(10, 20, "generic");

    for(int i=0; i < shapes.Length; i++) {
      Console.WriteLine("object is " + shapes[i].name);
      Console.WriteLine("Area is " + shapes[i].Area());

      Console.WriteLine();
    }
  }
}
```

The output from the program is shown here:

```
object is triangle
Area is 48

object is rectangle
Area is 100

object is rectangle
Area is 40

object is triangle
Area is 24.5

object is generic
Area() must be overridden
Area is 0
```

Let's examine this program closely. First, as explained, **Area( )** is declared as **virtual** in the **TwoDShape** class and is overridden by **Triangle** and **Rectangle**. Inside **TwoDShape**, **Area( )** is given a placeholder implementation that simply informs the user that this method must be overridden by a derived class. Each override of **Area( )** supplies an implementation that is suitable for the type of object encapsulated by the derived class. Thus, if you were to implement an ellipse class, for example, then **Area( )** would need to compute the area of an ellipse.

There is one other important feature in the preceding program. Notice in **Main( )** that **shapes** is declared as an array of **TwoDShape** objects. However, the elements of this array are assigned **Triangle**, **Rectangle**, and **TwoDShape** references. This is valid because a base class reference can refer to a derived class object. The program then cycles through the array, displaying information about each object. Although quite simple, this illustrates the power of both inheritance and method overriding. The type of object stored in a base class reference variable is determined at runtime and acted on accordingly. If an object is derived from **TwoDShape**, then its area can be obtained by calling **Area( )**. The interface to this operation is the same no matter what type of shape is being used.

Using Abstract Classes

Sometimes you will want to create a base class that defines only a generalized form that will be shared by all of its derived classes, leaving it to each derived class to fill in the details. Such a class determines the nature of the methods that the derived classes must implement, but does not, itself, provide an implementation of one or more of these methods. One way this situation can occur is when a base class is unable to create a meaningful implementation for a method. This is the case with the version of **TwoDShape** used in the preceding example. The definition of **Area( )** is simply a placeholder. It will not compute and display the area of any type of object.

You will see as you create your own class libraries that it is not uncommon for a method to have no meaningful definition in the context of its base class. You can handle this situation two ways. One way, as shown in the previous example, is to simply have it report a warning message. Although this approach can be useful in certain situations—such as debugging—it is not usually appropriate. You may have methods that must be overridden by the derived class in order for the derived class to have any meaning. Consider the class **Triangle**. It is incomplete if **Area( )** is not defined. In such a case, you want some way to ensure that a derived class does, indeed, override all necessary methods. C#'s solution to this problem is the *abstract method.*

An abstract method is created by specifying the **abstract** type modifier. An abstract method contains no body and is, therefore, not implemented by the base class. Thus, a derived class must override it—it cannot simply use the version defined in the base class. As you can probably guess, an abstract method is automatically virtual, and there is no need to use the **virtual** modifier. In fact, it is an error to use **virtual** and **abstract** together.

To declare an abstract method, use this general form:

abstract *type name*(*parameter-list*);

As you can see, no method body is present. The **abstract** modifier can be used only on instance methods. It cannot be applied to **static** methods. Properties and indexers can also be abstract.

A class that contains one or more abstract methods must also be declared as abstract by preceding its **class** declaration with the **abstract** specifier. Since an abstract class does not define a complete implementation, there can be no objects of an abstract class. Thus, attempting to create an object of an abstract class by using **new** will result in a compile-time error.

When a derived class inherits an abstract class, it must implement all of the abstract methods in the base class. If it doesn't, then the derived class must also be specified as **abstract**. Thus, the **abstract** attribute is inherited until such time as a complete implementation is achieved.

Using an abstract class, you can improve the **TwoDShape** class. Since there is no meaningful concept of area for an undefined two-dimensional figure, the following version of the preceding program declares **Area( )** as **abstract** inside **TwoDShape** and **TwoDShape** as **abstract**. This, of course, means that all classes derived from **TwoDShape** must override **Area( )**.

```
// Create an abstract class.

using System;

abstract class TwoDShape {
  double pri_width;
  double pri_height;

  // A default constructor.
  public TwoDShape() {
    Width = Height = 0.0;
    name = "null";
  }

  // Parameterized constructor.
  public TwoDShape(double w, double h, string n) {
    Width = w;
    Height = h;
    name = n;
  }

  // Construct object with equal width and height.
  public TwoDShape(double x, string n) {
    Width = Height = x;
    name = n;
  }

  // Construct a copy of a TwoDShape object.
  public TwoDShape(TwoDShape ob) {
    Width = ob.Width;
    Height = ob.Height;
    name = ob.name;
  }

  // Properties for Width and Height.
  public double Width {
     get { return pri_width; }
     set { pri_width = value < 0 ? -value : value; }
  }
```

```
  public double Height {
     get { return pri_height; }
     set { pri_height = value < 0 ? -value : value; }
  }

  public string name { get; set; }

  public void ShowDim() {
    Console.WriteLine("Width and height are " +
                      Width + " and " + Height);
  }

  // Now, Area() is abstract.
  public abstract double Area();
}

// A derived class of TwoDShape for triangles.
class Triangle : TwoDShape {
  string Style;

  // A default constructor.
  public Triangle() {
    Style = "null";
  }

  // Constructor for Triangle.
  public Triangle(string s, double w, double h) :
    base(w, h, "triangle") {
      Style = s;
  }

  // Construct an isosceles triangle.
  public Triangle(double x) : base(x, "triangle") {
    Style = "isosceles";
  }

  // Construct a copy of a Triangle object.
  public Triangle(Triangle ob) : base(ob) {
    Style = ob.Style;
  }

  // Override Area() for Triangle.
  public override double Area() {
    return Width * Height / 2;
  }

  // Display a triangle's style.
  public void ShowStyle() {
    Console.WriteLine("Triangle is " + Style);
  }
}

// A derived class of TwoDShape for rectangles.
class Rectangle : TwoDShape {
```

```
  // Constructor for Rectangle.
  public Rectangle(double w, double h) :
    base(w, h, "rectangle"){ }

  // Construct a square.
  public Rectangle(double x) :
    base(x, "rectangle") { }

  // Construct a copy of a Rectangle object.
  public Rectangle(Rectangle ob) : base(ob) { }

  // Return true if the rectangle is square.
  public bool IsSquare() {
    if(Width == Height) return true;
    return false;
  }

  // Override Area() for Rectangle.
  public override double Area() {
    return Width * Height;
  }
}

class AbsShape {
  static void Main() {
    TwoDShape[] shapes = new TwoDShape[4];

    shapes[0] = new Triangle("right", 8.0, 12.0);
    shapes[1] = new Rectangle(10);
    shapes[2] = new Rectangle(10, 4);
    shapes[3] = new Triangle(7.0);

    for(int i=0; i < shapes.Length; i++) {
      Console.WriteLine("object is " + shapes[i].name);
      Console.WriteLine("Area is " + shapes[i].Area());

      Console.WriteLine();
    }
  }
}
```

As the program illustrates, all derived classes *must* override **Area( )** (or also be declared **abstract**). To prove this to yourself, try creating a derived class that does not override **Area( )**. You will receive a compile-time error. Of course, it is still possible to create an object reference of type **TwoDShape**, which the program does. However, it is no longer possible to declare objects of type **TwoDShape**. Because of this, in **Main( )** the **shapes** array has been shortened to 4, and a generic **TwoDShape** object is no longer created.

One other point: Notice that **TwoDShape** still includes the **ShowDim( )** method and that it is not modified by **abstract**. It is perfectly acceptable—indeed, quite common—for an abstract class to contain concrete methods that a derived class is free to use as-is. Only those methods declared as **abstract** must be overridden by derived classes.

Using sealed to Prevent Inheritance

As powerful and useful as inheritance is, sometimes you will want to prevent it. For example, you might have a class that encapsulates the initialization sequence of some specialized hardware device, such as a medical monitor. In this case, you don't want users of your class to be able to change the way the monitor is initialized, possibly setting the device incorrectly. Whatever the reason, in C# it is easy to prevent a class from being inherited by using the keyword **sealed**.

To prevent a class from being inherited, precede its declaration with **sealed**. As you might expect, it is illegal to declare a class as both **abstract** and **sealed** because an abstract class is incomplete by itself and relies upon its derived classes to provide complete implementations.

Here is an example of a **sealed** class:

```
sealed class A {
  // ...
}

// The following class is illegal.
class B : A { // ERROR! Can't derive from class A
  // ...
}
```

As the comments imply, it is illegal for **B** to inherit **A** because **A** is declared as **sealed**.

One other point: **sealed** can also be used on virtual methods to prevent further overrrides. For example, assume a base class called **B** and a derived class called **D**. A method declared **virtual** in **B** can be declared **sealed** by **D**. This would prevent any class that inherits **D** from overriding the method. This situation is illustrated by the following:

```
class B {
  public virtual void MyMethod() { /* ... */ }
}

class D : B {
  // This seals MyMethod() and prevents further overrides.
  sealed public override void MyMethod() { /* ... */ }
}

class X : D {
  // Error! MyMethod() is sealed!
  public override void MyMethod() { /* ... */ }
}
```

Because **MyMethod()** is sealed by **D**, it can't be overridden by **X**.

The object Class

C# defines one special class called **object** that is an implicit base class of all other classes and for all other types (including the value types). In other words, all other types are derived from **object**. This means that a reference variable of type **object** can refer to an object of any

other type. Also, since arrays are implemented as objects, a variable of type **object** can also refer to any array. Technically, the C# name **object** is just another name for **System.Object**, which is part of the .NET Framework class library.

The **object** class defines the methods shown in Table 11-1, which means that they are available in every object.

A few of these methods warrant some additional explanation. By default, the **Equals(object)** method determines if the invoking object refers to the same object as the one referred to by the argument. (That is, it determines if the two references are the same.) It returns **true** if the objects are the same, and **false** otherwise. You can override this method in classes that you create. Doing so allows you to define what equality means relative to a class. For example, you could define **Equals(object)** so that it compares the contents of two objects for equality. The **Equals(object, object)** method invokes **Equals(object)** to compute its result.

The **GetHashCode( )** method returns a hash code associated with the invoking object. This hash code can be used with any algorithm that employs hashing as a means of accessing stored objects.

As mentioned in Chapter 9, if you overload the **= =** operator, then you will usually need to override **Equals(object)** and **GetHashCode( )** because most of the time you will want the **= =** operator and the **Equals(object)** methods to function the same. When **Equals( )** is overridden, you should also override **GetHashCode( )**, so that the two methods are compatible.

The **ToString( )** method returns a string that contains a description of the object on which it is called. Also, this method is automatically called when an object is output using

Method	Purpose
public virtual bool Equals(object *ob*)	Determines whether the invoking object is the same as the one referred to by *ob.*
public static bool Equals(object *ob1*, object *ob2*)	Determines whether *ob1* is the same as *ob2.*
protected virtual Finalize( )	Performs shutdown actions prior to garbage collection. In C#, **Finalize( )** is accessed through a destructor.
public virtual int GetHashCode( )	Returns the hash code associated with the invoking object.
public Type GetType( )	Obtains the type of an object at runtime.
protected object MemberwiseClone( )	Makes a "shallow copy" of the object. This is one in which the members are copied, but objects referred to by members are not.
public static bool ReferenceEquals(object *ob1*, object *ob2*)	Determines whether *ob1* and *ob2* refer to the same object.
public virtual string ToString( )	Returns a string that describes the object.

TABLE 11-1 Methods of the **object** Class

WriteLine( ). Many classes override this method. Doing so allows them to tailor a description specifically for the types of objects that they create. For example:

```
// Demonstrate ToString()

using System;

class MyClass {
  static int count = 0;
  int id;

  public MyClass() {
    id = count;
    count++;
  }

  public override string ToString() {
    return "MyClass object #" + id;
  }
}

class Test {
  static void Main() {
    MyClass ob1 = new MyClass();
    MyClass ob2 = new MyClass();
    MyClass ob3 = new MyClass();

    Console.WriteLine(ob1);
    Console.WriteLine(ob2);
    Console.WriteLine(ob3);
  }
}
```

The output from the program is shown here:

```
MyClass object #0
MyClass object #1
MyClass object #2
```

Boxing and Unboxing

As explained, all C# types, including the value types, are derived from **object**. Thus, a reference of type **object** can be used to refer to any other type, including value types. When an **object** reference refers to a value type, a process known as *boxing* occurs. Boxing causes the value of a value type to be stored in an object instance. Thus, a value type is "boxed" inside an object. This object can then be used like any other object. In all cases, boxing occurs automatically. You simply assign a value to an **object** reference. C# handles the rest.

Unboxing is the process of retrieving a value from a boxed object. This action is performed using an explicit cast from the **object** reference to its corresponding value type. Attempting to unbox an object into a different type will result in a runtime error.

Here is a simple example that illustrates boxing and unboxing:

```
// A simple boxing/unboxing example.

using System;

class BoxingDemo {
  static void Main() {
    int x;
    object obj;

    x = 10;
    obj = x; // box x into an object

    int y = (int)obj; // unbox obj into an int
    Console.WriteLine(y);
  }
}
```

This program displays the value 10. Notice that the value in **x** is boxed simply by assigning it to **obj**, which is an **object** reference. The integer value in **obj** is retrieved by casting **obj** to **int**.

Here is another, more interesting example of boxing. In this case, an **int** is passed as an argument to the **Sqr( )** method, which uses an **object** parameter.

```
// Boxing also occurs when passing values.

using System;

class BoxingDemo {
  static void Main() {
    int x;

    x = 10;
    Console.WriteLine("Here is x: " + x);

    // x is automatically boxed when passed to Sqr().
    x = BoxingDemo.Sqr(x);
    Console.WriteLine("Here is x squared: " + x);
  }

  static int Sqr(object o) {
    return (int)o * (int)o;
  }
}
```

The output from the program is shown here:

```
Here is x: 10
Here is x squared: 100
```

Here, the value of **x** is automatically boxed when it is passed to **Sqr( )**.

Boxing and unboxing allow C#'s type system to be fully unified. All types derive from **object**. A reference to any type can be assigned to a variable of type **object**. Boxing and

unboxing automatically handle the details for the value types. Furthermore, because all types are derived from **object**, they all have access to **object**'s methods. For example, consider the following rather surprising program:

```
// Boxing makes it possible to call methods on a value!

using System;

class MethOnValue {
  static void Main() {

    Console.WriteLine(10.ToString());

  }
}
```

This program displays 10. The reason is that the **ToString( )** method returns a string representation of the object on which it is called. In this case, the string representation of 10 is 10!

Is object a Universal Data Type?

Given that **object** is a base class for all other types and that boxing of the value types takes place automatically, it is possible to use **object** as a "universal" data type. For example, consider the following program that creates an array of **object** and then assigns various other types of data to its elements:

```
// Use object to create a "generic" array.

using System;

class GenericDemo {
  static void Main() {
    object[] ga = new object[10];

    // Store ints.
    for(int i=0; i < 3; i++)
      ga[i] = i;

    // Store doubles.
    for(int i=3; i < 6; i++)
      ga[i] = (double) i / 2;


    // Store two strings, a bool, and a char.
    ga[6] = "Hello";
    ga[7] = true;
    ga[8] = 'X';
    ga[9] = "end";

    for(int i = 0; i < ga.Length; i++)
      Console.WriteLine("ga[" + i + "]: " + ga[i] + " ");
  }
}
```

The output is shown here:

```
ga[0]: 0
ga[1]: 1
ga[2]: 2
ga[3]: 1.5
ga[4]: 2
ga[5]: 2.5
ga[6]: Hello
ga[7]: True
ga[8]: X
ga[9]: end
```

As this program illustrates, because an **object** reference can hold a reference to any other type of data, it is possible to use an **object** reference to refer to any type of data. Thus, an array of **object** as used by the program can store any type of data. Expanding on this concept, it is easy to see how you could construct a stack class, for example, that stored **object** references. This would enable the stack to store any type of data.

Although the universal-type feature of **object** is powerful and can be used quite effectively in some situations, it is a mistake to think that you should use **object** as a way around C#'s otherwise strong type checking. In general, when you need to store an **int**, use an **int** variable; when you need to store a **string**, use a **string** reference; and so on.

More importantly, since version 2.0, true generic types are available to the C# programmer. (Generics are described in Chapter 18.) The addition of generics enables you to easily define classes and algorithms that automatically work with different types of data in a type-safe manner. Because of generics, you will normally not need to use **object** as a universal type when creating new code. Today, it's best to reserve **object**'s universal nature for specialized situations.

12
CHAPTER

Interfaces, Structures, and Enumerations

This chapter discusses one of C#'s most important features: the interface. An *interface* defines a set of methods that will be implemented by a class. An interface does not, itself, implement any method. Thus, an interface is a purely logical construct that describes functionality without specifying implementation.

Also discussed in this chapter are two more C# data types: structures and enumerations. *Structures* are similar to classes except that they are handled as value types rather than reference types. *Enumerations* are lists of named integer constants. Structures and enumerations contribute to the richness of the C# programming environment.

Interfaces

In object-oriented programming it is sometimes helpful to define what a class must do, but not how it will do it. You have already seen an example of this: the abstract method. An abstract method declares the return type and signature for a method, but provides no implementation. A derived class must provide its own implementation of each abstract method defined by its base class. Thus, an abstract method specifies the *interface* to the method, but not the *implementation.* Although abstract classes and methods are useful, it is possible to take this concept a step further. In C#, you can fully separate a class' interface from its implementation by using the keyword **interface**.

Interfaces are syntactically similar to abstract classes. However, in an interface, no method can include a body. That is, an interface provides no implementation whatsoever. It specifies what must be done, but not how. Once an interface is defined, any number of classes can implement it. Also, one class can implement any number of interfaces.

To implement an interface, a class must provide bodies (implementations) for the methods described by the interface. Each class is free to determine the details of its own implementation. Thus, two classes might implement the same interface in different ways, but each class still supports the same set of methods. Therefore, code that has knowledge of the interface can use objects of either class since the interface to those objects is the same. By providing the interface, C# allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

Interfaces are declared by using the **interface** keyword. Here is a simplified form of an interface declaration:

```
interface name {
  ret-type method-name1(param-list);
  ret-type method-name2(param-list);
  // ...
  ret-type method-nameN(param-list);
}
```

The name of the interface is specified by *name.* Methods are declared using only their return type and signature. They are, essentially, abstract methods. As explained, in an interface, no method can have an implementation. Thus, each class that includes an interface must implement all of the methods. In an interface, methods are implicitly **public**, and no explicit access specifier is allowed.

Here is an example of an interface. It specifies the interface to a class that generates a series of numbers.

```
public interface ISeries {
  int GetNext(); // return next number in series
  void Reset(); // restart
  void SetStart(int x); // set starting value
}
```

The name of this interface is **ISeries**. Although the prefix **I** is not necessary, many programmers prefix interfaces with **I** to differentiate them from classes. **ISeries** is declared **public** so that it can be implemented by any class in any program.

In addition to methods, interfaces can specify properties, indexers, and events. Events are described in Chapter 15, and we will be concerned with only methods, properties, and indexers here. Interfaces cannot have data members. They cannot define constructors, destructors, or operator methods. Also, no member can be declared as **static**.

Implementing Interfaces

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, the name of the interface is specified after the class name in just the same way that a base class is specified. The general form of a class that implements an interface is shown here:

```
class class-name : interface-name {
  // class-body
}
```

The name of the interface being implemented is specified in *interface-name.* When a class implements an interface, the class must implement the entire interface. It cannot pick and choose which parts to implement, for example.

A class can implement more than one interface. When a class implements more than one interface, specify each interface in a comma-separated list. A class can inherit a base class and also implement one or more interfaces. In this case, the name of the base class must come first in the comma-separated list.

The methods that implement an interface must be declared **public**. The reason for this is that methods are implicitly public within an interface, so their implementation must also be public. Also, the return type and signature of the implementing method must match exactly the return type and signature specified in the **interface** definition.

Here is an example that implements the **ISeries** interface shown earlier. It creates a class called **ByTwos**, which generates a series of numbers, each two greater than the previous one.

```
// Implement ISeries.
class ByTwos : ISeries {
  int start;
  int val;

  public ByTwos() {
    start = 0;
    val = 0;
  }

  public int GetNext() {
    val += 2;
    return val;
  }

  public void Reset() {
    val = start;
  }

  public void SetStart(int x) {
    start = x;
    val = start;
  }
}
```

As you can see, **ByTwos** implements all three methods defined by **ISeries**. As explained, this is necessary since a class cannot create a partial implementation of an interface.

Here is a class that demonstrates **ByTwos**:

```
// Demonstrate the ByTwos interface.

using System;

class SeriesDemo {
  static void Main() {
    ByTwos ob = new ByTwos();

    for(int i=0; i < 5; i++)
      Console.WriteLine("Next value is " +
                        ob.GetNext());

    Console.WriteLine("\nResetting");
    ob.Reset();
    for(int i=0; i < 5; i++)
      Console.WriteLine("Next value is " +
                        ob.GetNext());
```

```
    Console.WriteLine("\nStarting at 100");
    ob.SetStart(100);
    for(int i=0; i < 5; i++)
      Console.WriteLine("Next value is " +
                        ob.GetNext());
  }
}
```

To compile **SeriesDemo**, you must include the files that contain **ISeries**, **ByTwos**, and **SeriesDemo** in the compilation. The compiler will automatically compile all three files to create the final executable. For example, if you called these files **ISeries.cs**, **ByTwos.cs**, and **SeriesDemo.cs**, then the following command line will compile the program:

```
>csc SeriesDemo.cs ISeries.cs ByTwos.cs
```

If you are using the Visual C++ IDE, simply add all three files to your C# project. One other point: It is perfectly valid to put all three of these classes in the same file, too.

The output from this program is shown here:

```
Next value is 2
Next value is 4
Next value is 6
Next value is 8
Next value is 10

Resetting
Next value is 2
Next value is 4
Next value is 6
Next value is 8
Next value is 10

Starting at 100
Next value is 102
Next value is 104
Next value is 106
Next value is 108
Next value is 110
```

It is both permissible and common for classes that implement interfaces to define additional members of their own. For example, the following version of **ByTwos** adds the method **GetPrevious( )**, which returns the previous value:

```
// Implement ISeries and add GetPrevious().
 class ByTwos : ISeries {
  int start;
  int val;
  int prev;

  public ByTwos() {
    start = 0;
    val = 0;
    prev = -2;
  }
```

```
  public int GetNext() {
    prev = val;
    val += 2;
    return val;
  }

  public void Reset() {
    val = start;
    prev = start - 2;
  }

  public void SetStart(int x) {
    start = x;
    val = start;
    prev = val - 2;
  }

  // A method not specified by ISeries.
  public int GetPrevious() {
    return prev;
  }
}
```

Notice that the addition of **GetPrevious( )** required a change to the implementations of the methods defined by **ISeries**. However, since the interface to those methods stays the same, the change is seamless and does not break preexisting code. This is one of the advantages of interfaces.

As explained, any number of classes can implement an **interface**. For example, here is a class called **Primes** that generates a series of prime numbers. Notice that its implementation of **ISeries** is fundamentally different than the one provided by **ByTwos**.

```
// Use ISeries to implement a series of prime numbers.
class Primes : ISeries {
  int start;
  int val;

  public Primes() {
    start = 2;
    val = 2;
  }

  public int GetNext() {
    int i, j;
    bool isprime;

    val++;
    for(i = val; i < 1000000; i++) {
      isprime = true;
      for(j = 2; j <= i/j; j++) {
        if((i%j)==0) {
          isprime = false;
          break;
        }
      }
```

```
      if(isprime) {
        val = i;
        break;
      }
    }
    return val;
  }

  public void Reset() {
    val = start;
  }

  public void SetStart(int x) {
    start = x;
    val = start;
  }
}
```

The key point is that even though **ByTwos** and **Primes** generate completely unrelated series of numbers, both implement **ISeries**. As explained, an interface says nothing about the implementation, so each class is free to implement the interface as it sees fit.

Using Interface References

You might be somewhat surprised to learn that you can declare a reference variable of an interface type. In other words, you can create an interface reference variable. Such a variable can refer to any object that implements its interface. When you call a method on an object through an interface reference, it is the version of the method implemented by the object that is executed. This process is similar to using a base class reference to access a derived class object, as described in Chapter 11.

The following example illustrates the use of an interface reference. It uses the same interface reference variable to call methods on objects of both **ByTwos** and **Primes**. For clarity, it shows all pieces of the program, assembled into a single file.

```
// Demonstrate interface references.

using System;

// Define the interface.
public interface ISeries {
  int GetNext(); // return next number in series
  void Reset(); // restart
  void SetStart(int x); // set starting value
}

// Use ISeries to implement a series in which each
// value is two greater than the previous one.
class ByTwos : ISeries {
  int start;
  int val;
```

```
  public ByTwos() {
    start = 0;
    val = 0;
  }

  public int GetNext() {
    val += 2;
    return val;
  }

  public void Reset() {
    val = start;
  }

  public void SetStart(int x) {
    start = x;
    val = start;
  }
}

// Use ISeries to implement a series of prime numbers.
class Primes : ISeries {
  int start;
  int val;

  public Primes() {
    start = 2;
    val = 2;
  }

  public int GetNext() {
    int i, j;
    bool isprime;

    val++;
    for(i = val; i < 1000000; i++) {
      isprime = true;
      for(j = 2; j <= i/j; j++) {
        if((i%j)==0) {
          isprime = false;
          break;
        }
      }
      if(isprime) {
        val = i;
        break;
      }
    }
    return val;
  }

  public void Reset() {
    val = start;
  }
```

```
  public void SetStart(int x) {
    start = x;
    val = start;
  }
}

class SeriesDemo2 {
  static void Main() {
    ByTwos twoOb = new ByTwos();
    Primes primeOb = new Primes();
    ISeries ob;

    for(int i=0; i < 5; i++) {
      ob = twoOb;
      Console.WriteLine("Next ByTwos value is " +
                        ob.GetNext());
      ob = primeOb;
      Console.WriteLine("Next prime number is " +
                        ob.GetNext());
    }
  }
}
```

The output from the program is shown here:

```
Next ByTwos value is 2
Next prime number is 3
Next ByTwos value is 4
Next prime number is 5
Next ByTwos value is 6
Next prime number is 7
Next ByTwos value is 8
Next prime number is 11
Next ByTwos value is 10
Next prime number is 13
```

In **Main( )**, **ob** is declared to be a reference to an **ISeries** interface. This means that it can be used to store references to any object that implements **ISeries**. In this case, it is used to refer to **twoOb** and **primeOb**, which are objects of type **ByTwos** and **Primes**, respectively, which both implement **ISeries**.

One other point: An interface reference variable has knowledge only of the methods declared by its **interface** declaration. Thus, an interface reference cannot be used to access any other variables or methods that might be supported by the object.

Interface Properties

Like methods, properties are specified in an interface without any body. Here is the general form of a property specification:

```
// interface property
type name {
  get;
  set;
}
```

Of course, only **get** or **set** will be present for read-only or write-only properties, respectively.

Although the declaration of a property in an interface looks similar to how an auto-implemented property is declared in a class, the two are not the same. The interface declaration does not cause the property to be auto-implemented. It only specifies the name and type of the property. Implementation is left to each implementing class. Also, no access modifiers are allowed on the accessors when a property is declared in an **interface**. Thus, the **set** accessor, for example, cannot be specified as private in an **interface**.

Here is a rewrite of the **ISeries** interface and the **ByTwos** class that uses a property called **Next** to obtain and set the next element in the series:

```
// Use a property in an interface.

using System;

public interface ISeries {
  // An interface property.
  int Next {
    get; // return the next number in series
    set; // set next number
  }
}

// Implement ISeries.
class ByTwos : ISeries {
  int val;

  public ByTwos() {
    val = 0;
  }

  // Get or set value.
  public int Next {
    get {
      val += 2;
      return val;
    }
    set {
      val = value;
    }
  }
}

// Demonstrate an interface property.
class SeriesDemo3 {
  static void Main() {
    ByTwos ob = new ByTwos();

    // Access series through a property.
    for(int i=0; i < 5; i++)
      Console.WriteLine("Next value is " + ob.Next);

    Console.WriteLine("\nStarting at 21");
    ob.Next = 21;
```

```
    for(int i=0; i < 5; i++)
      Console.WriteLine("Next value is " + ob.Next);
  }
}
```

The output from this program is shown here:

```
Next value is 2
Next value is 4
Next value is 6
Next value is 8
Next value is 10

Starting at 21
Next value is 23
Next value is 25
Next value is 27
Next value is 29
Next value is 31
```

Interface Indexers

An interface can specify an indexer. A one-dimensional indexer declared in an interface has this general form:

```
// interface indexer
element-type this[int index] {
  get;
  set;
}
```

As before, only **get** or **set** will be present for read-only or write-only indexers, respectively. Also, no access modifiers are allowed on the accessors when an indexer is declared in an **interface**.

Here is another version of **ISeries** that adds a read-only indexer that returns the *i*-th element in the series.

```
// Add an indexer in an interface.

using System;

public interface ISeries {
  // An interface property.
  int Next {
    get; // return the next number in series
    set; // set next number
  }

  // An interface indexer.
  int this[int index] {
    get; // return the specified number in series
  }
}
```

```
// Implement ISeries.
class ByTwos : ISeries {
  int val;

  public ByTwos() {
    val = 0;
  }

  // Get or set value using a property.
  public int Next {
    get {
      val += 2;
      return val;
    }
    set {
      val = value;
    }
  }

  // Get a value using an index.
  public int this[int index] {
    get {
      val = 0;
      for(int i=0; i < index; i++)
        val += 2;
      return val;
    }
  }
}

// Demonstrate an interface indexer.
class SeriesDemo4 {
  static void Main() {
    ByTwos ob = new ByTwos();

    // Access series through a property.
    for(int i=0; i < 5; i++)
      Console.WriteLine("Next value is " + ob.Next);

    Console.WriteLine("\nStarting at 21");
    ob.Next = 21;
    for(int i=0; i < 5; i++)
      Console.WriteLine("Next value is " +
                        ob.Next);

    Console.WriteLine("\nResetting to 0");
    ob.Next = 0;

    // Access series through an indexer.
    for(int i=0; i < 5; i++)
      Console.WriteLine("Next value is " + ob[i]);
  }
}
```

The output from this program is shown here:

```
Next value is 2
Next value is 4
Next value is 6
Next value is 8
Next value is 10

Starting at 21
Next value is 23
Next value is 25
Next value is 27
Next value is 29
Next value is 31

Resetting to 0
Next value is 0
Next value is 2
Next value is 4
Next value is 6
Next value is 8
```

Interfaces Can Be Inherited

One interface can inherit another. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all the members defined within the interface inheritance chain. Here is an example:

```
// One interface can inherit another.

using System;

public interface IA {
  void Meth1();
  void Meth2();
}

// B now includes Meth1() and Meth2() -- it adds Meth3().
public interface IB : IA {
  void Meth3();
}

// This class must implement all of IA and IB.
class MyClass : IB {
  public void Meth1() {
    Console.WriteLine("Implement Meth1().");
  }

  public void Meth2() {
    Console.WriteLine("Implement Meth2().");
  }

  public void Meth3() {
    Console.WriteLine("Implement Meth3().");
```

```
  }
}

class IFExtend {
  static void Main() {
    MyClass ob = new MyClass();

    ob.Meth1();
    ob.Meth2();
    ob.Meth3();
  }
}
```

As an experiment, you might try removing the implementation for **Meth1( )** in **MyClass**. This will cause a compile-time error. As stated earlier, any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces.

Name Hiding with Interface Inheritance

When one interface inherits another, it is possible to declare a member in the derived interface that hides one defined by the base interface. This happens when a member in a derived interface has the same declaration as one in the base interface. In this case, the base interface name is hidden. This will cause a warning message unless you specify the derived interface member with **new**.

Explicit Implementations

When implementing a member of an interface, it is possible to *fully qualify* its name with its interface name. Doing this creates an *explicit interface member implementation*, or *explicit implementation*, for short. For example, given

```
interface IMyIF {
  int MyMeth(int x);
}
```

then it is legal to implement **IMyIF** as shown here:

```
class MyClass : IMyIF {
  int IMyIF.MyMeth(int x) {
    return x / 3;
  }
}
```

As you can see, when the **MyMeth( )** member of **IMyIF** is implemented, its complete name, including its interface name, is specified.

There are two reasons that you might need to create an explicit implementation of an interface method. First, when you implement an interface method using its fully qualified name, you are providing an implementation that *cannot* be accessed through an object of the class. Instead, it must be accessed via an interface reference. Thus, an explicit implementation gives you a way to implement an interface method so that it is not a public member of the

class that provides the implementation. Second, it is possible for a class to implement two interfaces, both of which declare methods by the same name and type signature. Qualifying the names with their interfaces removes the ambiguity from this situation. Let's look at an example of each.

The following program contains an interface called **IEven**, which defines two methods, **IsEven( )** and **IsOdd( )**, which determine if a number is even or odd. **MyClass** then implements **IEven**. When it does so, it implements **IsOdd( )** explicitly.

```
// Explicitly implement an interface member.

using System;

interface IEven {
  bool IsOdd(int x);
  bool IsEven(int x);
}

class MyClass : IEven {

  // Explicit implementation. Notice that this member is private
  // by default.
  bool IEven.IsOdd(int x) {
    if((x%2) != 0) return true;
    else return false;
  }

  // Normal implementation.
  public bool IsEven(int x) {
    IEven o = this; // Interface reference to the invoking object.

    return !o.IsOdd(x);
  }
}

class Demo {
  static void Main() {
    MyClass ob = new MyClass();
    bool result;

    result = ob.IsEven(4);
    if(result) Console.WriteLine("4 is even.");

    // result = ob.IsOdd(4); // Error, IsOdd not exposed.

    // But, this is OK. It creates an IEven reference to a MyClass object
    // and then calls IsOdd() through that reference.
    IEven iRef = (IEven) ob;
    result = iRef.IsOdd(3);
    if(result) Console.WriteLine("3 is odd.");

  }
}
```

Since **IsOdd()** is implemented explicitly, it is not exposed as a public member of **MyClass**. Instead, **IsOdd()** can be accessed only through an interface reference. This is why it is invoked through **o** (which is a reference variable of type **IEven**) in the implementation for **IsEven()**.

Here is an example in which two interfaces are implemented and both interfaces declare a method called **Meth( )**. Explicit implementation is used to eliminate the ambiguity inherent in this situation.

```
// Use explicit implementation to remove ambiguity.

using System;

interface IMyIF_A {
  int Meth(int x);
}

interface IMyIF_B {
  int Meth(int x);
}

// MyClass implements both interfaces.
class MyClass : IMyIF_A, IMyIF_B {

  // Explicitly implement the two Meth()s.
  int IMyIF_A.Meth(int x) {
    return x + x;
  }
  int IMyIF_B.Meth(int x) {
    return x * x;
  }

  // Call Meth() through an interface reference.
  public int MethA(int x){
    IMyIF_A a_ob;
    a_ob = this;
    return a_ob.Meth(x); // calls IMyIF_A
  }

  public int MethB(int x){
    IMyIF_B b_ob;
    b_ob = this;
    return b_ob.Meth(x); // calls IMyIF_B
  }
}

class FQIFNames {
  static void Main() {
    MyClass ob = new MyClass();

    Console.Write("Calling IMyIF_A.Meth(): ");
    Console.WriteLine(ob.MethA(3));
```

```
    Console.Write("Calling IMyIF_B.Meth(): ");
    Console.WriteLine(ob.MethB(3));
  }
}
```

The output from this program is shown here:

```
Calling IMyIF_A.Meth(): 6
Calling IMyIF_B.Meth(): 9
```

Looking at the program, first notice that **Meth()** has the same signature in both **IMyIF_A** and **IMyIF_B**. Thus, when **MyClass** implements both of these interfaces, it explicitly implements each one separately, fully qualifying its name in the process. Since the only way that an explicitly implemented method can be called is on an interface reference, **MyClass** creates two such references, one for **IMyIF_A** and one for **IMyIF_B**. It then calls two of its own methods, which call the interface methods, thereby removing the ambiguity.

Choosing Between an Interface and an Abstract Class

One of the more challenging parts of C# programming is knowing when to create an interface and when to use an abstract class in cases in which you want to describe functionality but not implementation. The general rule is this: When you can fully describe the concept in terms of "what it does" without needing to specify any "how it does it," then you should use an interface. If you need to include some implementation details, then you will need to represent your concept in an abstract class.

The .NET Standard Interfaces

The .NET Framework defines a large number of interfaces that a C# program can use. For example, **System.IComparable** defines the **CompareTo( )** method, which allows objects to be compared when an ordering relationship is required. Interfaces also form an important part of the Collections classes, which provide various types of storage (such as stacks and queues) for groups of objects. For example, **System.Collections.ICollection** defines the functionality of a collection. **System.Collections.IEnumerator** offers a way to sequence through the elements in a collection. These and many other interfaces are described in Part II.

Structures

As you know, classes are reference types. This means that class objects are accessed through a reference. This differs from the value types, which are accessed directly. However, sometimes it would be useful to be able to access an object directly, in the way that value types are. One reason for this is efficiency. Accessing class objects through a reference adds overhead onto every access. It also consumes space. For very small objects, this extra space might be significant. To address these concerns, C# offers the structure. A *structure* is similar to a class, but is a value type, rather than a reference type.

Structures are declared using the keyword **struct** and are syntactically similar to classes. Here is the general form of a **struct**:

```
struct name : interfaces {
  // member declarations
}
```

The name of the structure is specified by *name*.

Structures cannot inherit other structures or classes or be used as a base for other structures or classes. (Of course, like all C# types, structures do inherit **object**.) However, a structure can implement one or more interfaces. These are specified after the structure name using a comma-separated list. Like classes, structure members include methods, fields, indexers, properties, operator methods, and events. Structures can also define constructors, but not destructors. However, you cannot define a default (parameterless) constructor for a structure. The reason for this is that a default constructor is automatically defined for all structures, and this default constructor can't be changed. The default constructor initializes the fields of a structure to their default value. Since structures do not support inheritance, structure members cannot be specified as **abstract**, **virtual**, or **protected**.

A structure object can be created using **new** in the same way as a class object, but it is not required. When **new** is used, the specified constructor is called. When **new** is not used, the object is still created, but it is not initialized. Thus, you will need to perform any initialization manually.

Here is an example that uses a structure to hold information about a book:

```
// Demonstrate a structure.

using System;

// Define a structure.
struct Book {
  public string Author;
  public string Title;
  public int Copyright;

  public Book(string a, string t, int c) {
    Author = a;
    Title = t;
    Copyright = c;
  }
}

// Demonstrate Book structure.
class StructDemo {
  static void Main() {
    Book book1 = new Book("Herb Schildt",
                          "C# 3.0: The Complete Reference",
                          2009); // explicit constructor

    Book book2 = new Book(); // default constructor
    Book book3; // no constructor

    Console.WriteLine(book1.Title + " by " + book1.Author +
                      ", (c) " + book1.Copyright);
    Console.WriteLine();
```

```
    if(book2.Title == null)
      Console.WriteLine("book2.Title is null.");

    // Now, give book2 some info.
    book2.Title = "Brave New World";
    book2.Author = "Aldous Huxley";
    book2.Copyright = 1932;
    Console.Write("book2 now contains: ");
    Console.WriteLine(book2.Title + " by " + book2.Author +
                      ", (c) " + book2.Copyright);

    Console.WriteLine();

// Console.WriteLine(book3.Title); // error, must initialize first
    book3.Title = "Red Storm Rising";

    Console.WriteLine(book3.Title); // now OK
  }
}
```

The output from this program is shown here:

```
C# 3.0: The Complete Reference by Herb Schildt, (c) 2009

book2.Title is null.
book2 now contains: Brave New World by Aldous Huxley, (c) 1932

Red Storm Rising
```

As the program shows, a structure can be initialized either by using **new** to invoke a constructor or by simply declaring an object. If **new** is used, then the fields of the structure will be initialized either by the default constructor, which initializes all fields to their default value, or by a user-defined constructor. If **new** is not used, as is the case with **book3**, then the object is not initialized, and its fields must be set prior to using the object.

When you assign one structure to another, a copy of the object is made. This is an important way in which **struct** differs from **class**. As explained earlier in this book, when you assign one class reference to another, you are simply making the reference on the left side of the assignment refer to the same object as that referred to by the reference on the right. When you assign one **struct** variable to another, you are making a *copy* of the object on the right. For example, consider the following program:

```
// Copy a struct.

using System;

// Define a structure.
struct MyStruct {
  public int x;
}

// Demonstrate structure assignment.
class StructAssignment {
```

```
  static void Main() {
    MyStruct a;
    MyStruct b;

    a.x = 10;
    b.x = 20;

    Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);

    a = b;
    b.x = 30;

    Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);
  }
}
```

The output is shown here:

```
a.x 10, b.x 20
a.x 20, b.x 30
```

As the output shows, after the assignment

```
a = b;
```

the structure variables **a** and **b** are still separate and distinct. That is, **a** does not refer to or relate to **b** in any way other than containing a copy of **b**'s value. This would not be the case if **a** and **b** were class references. For example, here is the **class** version of the preceding program:

```
// Use a class.

using System;

// Now a class.
class MyClass {
  public int x;
}

// Now show a class object assignment.
class ClassAssignment {
  static void Main() {
    MyClass a = new MyClass();
    MyClass b = new MyClass();

    a.x = 10;
    b.x = 20;

    Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);

    a = b;
    b.x = 30;
```

```
    Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);
  }
}
```

The output from this version is shown here:

```
a.x 10, b.x 20
a.x 30, b.x 30
```

As you can see, after the assignment of **b** to **a**, both variables refer to the same object—the one originally referred to by **b**.

Why Structures?

At this point, you might be wondering why C# includes the **struct** since it seems to be a less-capable version of a **class**. The answer lies in efficiency and performance. Because structures are value types, they are operated on directly rather than through a reference. Thus, a **struct** does not require a separate reference variable. This means that less memory is used in some cases. Furthermore, because a **struct** is accessed directly, it does not suffer from the performance loss that is inherent in accessing a class object. Because classes are reference types, all access to class objects is through a reference. This indirection adds overhead to every access. Structures do not incur this overhead. In general, if you need to simply store a group of related data, but don't need inheritance and don't need to operate on that data through a reference, then a **struct** can be a more efficient choice.

Here is another example that shows how a structure might be used in practice. It simulates an e-commerce transaction record. Each transaction includes a packet header that contains the packet number and the length of the packet. This is followed by the account number and the amount of the transaction. Because the packet header is a self-contained unit of information, it is organized as a structure. This structure can then be used to create a transaction record, or any other type of information packet.

```
// Structures are good when grouping small amounts of data.

using System;

// Define a packet structure.
struct PacketHeader {
  public uint PackNum; // packet number
  public ushort PackLen; // length of packet
}

// Use PacketHeader to create an e-commerce transaction record.
class Transaction {
  static uint transacNum = 0;

  PacketHeader ph;  // incorporate PacketHeader into Transaction
  string accountNum;
  double amount;
```

```
  public Transaction(string acc, double val) {
   // create packet header
    ph.PackNum = transacNum++;
    ph.PackLen = 512;  // arbitrary length

    accountNum = acc;
    amount = val;
  }

  // Simulate a transaction.
  public void sendTransaction() {
    Console.WriteLine("Packet #: " + ph.PackNum +
                      ", Length: " + ph.PackLen +
                      ",\n    Account #: " + accountNum +
                      ", Amount: {0:C}\n", amount);
  }
}

// Demonstrate Packet.
class PacketDemo {
  static void Main() {
    Transaction t = new Transaction("31243", -100.12);
    Transaction t2 = new Transaction("AB4655", 345.25);
    Transaction t3 = new Transaction("8475-09", 9800.00);

    t.sendTransaction();
    t2.sendTransaction();
    t3.sendTransaction();
  }
}
```

The output from the program is shown here:

```
Packet #: 0, Length: 512,
    Account #: 31243, Amount: ($100.12)

Packet #: 1, Length: 512,
    Account #: AB4655, Amount: $345.25

Packet #: 2, Length: 512,
    Account #: 8475-09, Amount: $9,800.00
```

PacketHeader is a good choice for a **struct** because it contains only a small amount of data and does not use inheritance or even contain methods. As a structure, **PacketHeader** does not incur the additional overhead of a reference, as a class would. Thus, any type of transaction record can use **PacketHeader** without affecting its efficiency.

As a point of interest, C++ also has structures and uses the **struct** keyword. However, C# and C++ structures are not the same. In C++, **struct** defines a class type. Thus, in C++, **struct** and **class** are nearly equivalent. (The difference has to do with the default access of their members, which is private for **class** and public for **struct**.) In C#, a **struct** defines a value type, and a **class** defines a reference type.

Enumerations

An *enumeration* is a set of named integer constants. The keyword **enum** declares an enumerated type. The general form for an enumeration is

enum *name* { *enumeration list* };

Here, the type name of the enumeration is specified by *name*. The *enumeration list* is a comma-separated list of identifiers.

Here is an example. It defines an enumeration called **Apple** that enumerates various types of apples:

```
enum Apple { Jonathan, GoldenDel, RedDel, Winesap,
             Cortland, McIntosh };
```

A key point to understand about an enumeration is that each of the symbols stands for an integer value. However, no implicit conversions are defined between an **enum** type and the built-in integer types, so an explicit cast must be used. Also, a cast is required when converting between two enumeration types. Since enumerations represent integer values, you can use an enumeration to control a **switch** statement or as the control variable in a **for** loop, for example.

Each enumeration symbol is given a value one greater than the symbol that precedes it. By default, the value of the first enumeration symbol is 0. Therefore, in the **Apple** enumeration, **Jonathan** is 0, **GoldenDel** is 1, **RedDel** is 2, and so on.

The members of an enumeration are accessed through their type name via the dot operator. For example

```
Console.WriteLine(Apple.RedDel + " has the value " +
                  (int)Apple.RedDel);
```

displays

```
RedDel has the value 2
```

As the output shows, when an enumerated value is displayed, its name is used. To obtain its integer value, a cast to **int** must be employed.

Here is a program that illustrates the **Apple** enumeration:

```
// Demonstrate an enumeration.

using System;

class EnumDemo {
  enum Apple { Jonathan, GoldenDel, RedDel, Winesap,
               Cortland, McIntosh };

  static void Main() {
    string[] color = {
      "Red",
      "Yellow",
```

```
      "Red",
      "Red",
      "Red",
      "Reddish Green"
    };

    Apple i; // declare an enum variable

    // Use i to cycle through the enum.
    for(i = Apple.Jonathan; i <= Apple.McIntosh; i++)
      Console.WriteLine(i + " has value of " + (int)i);

    Console.WriteLine();

    // Use an enumeration to index an array.
    for(i = Apple.Jonathan; i <= Apple.McIntosh; i++)
      Console.WriteLine("Color of " + i + " is " +
                        color[(int)i]);
  }
}
```

The output from the program is shown here:

```
Jonathan has value of 0
GoldenDel has value of 1
RedDel has value of 2
Winesap has value of 3
Cortland has value of 4
McIntosh has value of 5

Color of Jonathan is Red
Color of GoldenDel is Yellow
Color of RedDel is Red
Color of Winesap is Red
Color of Cortland is Red
Color of McIntosh is Reddish Green
```

Notice how the **for** loops are controlled by a variable of type **Apple**. Because the enumerated values in **Apple** start at zero, these values can be used to index **color** to obtain the color of the apple. Notice that a cast is required when the enumeration value is used to index the **color** array. As mentioned, there are no implicit conversions defined between integers and enumeration types. An explicit cast is required.

Initialize an Enumeration

You can specify the value of one or more of the symbols by using an initializer. Do this by following the symbol with an equal sign and an integer value. Symbols that appear after initializers are assigned values greater than the previous initialization value. For example, the following code assigns the value of 10 to **RedDel**:

```
enum Apple { Jonathan, GoldenDel, RedDel = 10, Winesap,
             Cortland, McIntosh };
```

Now the values of these symbols are

Jonathan	0
GoldenDel	1
RedDel	10
Winesap	11
Cortland	12
McIntosh	13

Specify the Underlying Type of an Enumeration

By default, enumerations are based on type **int**, but you can create an enumeration of any integral type, except for type **char**. To specify a type other than **int**, put the desired type after the enumeration name, separated by a colon. For example, this statement makes **Apple** an enumeration based on **byte**:

```
enum Apple : byte { Jonathan, GoldenDel, RedDel, Winesap,
                    Cortland, McIntosh };
```

Now **Apple.Winesap**, for example, is a **byte** quantity.

Use Enumerations

At first glance you might think that enumerations are an interesting but relatively unimportant part of C#, yet this is not the case. Enumerations are very useful when your program requires one or more specialized symbols. For example, imagine that you are writing a program that controls a conveyor belt in a factory. You might create a method called **Conveyor( )** that accepts the following commands as parameters: start, stop, forward, and reverse. Instead of passing **Conveyor( )** integers, such as 1 for start, 2 for stop, and so on, which is error-prone, you can create an enumeration that assigns words to these values. Here is an example of this approach:

```
// Simulate a conveyor belt.

using System;

class ConveyorControl {
  // Enumerate the conveyor commands.
  public enum Action { Start, Stop, Forward, Reverse };

  public void Conveyor(Action com) {
    switch(com) {
      case Action.Start:
        Console.WriteLine("Starting conveyor.");
        break;
      case Action.Stop:
        Console.WriteLine("Stopping conveyor.");
        break;
```

```
      case Action.Forward:
        Console.WriteLine("Moving forward.");
        break;
      case Action.Reverse:
        Console.WriteLine("Moving backward.");
        break;
    }
  }
}

class ConveyorDemo {
  static void Main() {
    ConveyorControl c = new ConveyorControl();

    c.Conveyor(ConveyorControl.Action.Start);
    c.Conveyor(ConveyorControl.Action.Forward);
    c.Conveyor(ConveyorControl.Action.Reverse);
    c.Conveyor(ConveyorControl.Action.Stop);

  }
}
```

The output from the program is shown here:

```
Starting conveyor.
Moving forward.
Moving backward.
Stopping conveyor.
```

Because **Conveyor( )** takes an argument of type **Action**, only the values defined by **Action** can be passed to the method. For example, here an attempt is made to pass the value 22 to **Conveyor( )**:

```
c.Conveyor(22); // Error!
```

This won't compile because there is no predefined conversion from **int** to **Action**. This prevents the passing of invalid commands to **Conveyor( )**. Of course, you could use a cast to force a conversion, but this would require a premeditated act, not an accidental misuse. Also, because commands are specified by name rather than by number, it is less likely that a user of **Conveyor( )** will inadvertently pass the wrong value.

There is one other interesting thing in this example: Notice that an enumeration type is used to control the **switch** statement. As mentioned, because enumerations are integral types, they are perfectly valid for use in a **switch**.

This page intentionally left blank

CHAPTER 13

Exception Handling

An *exception* is an error that occurs at runtime. Using C#'s exception handling subsystem, you can, in a structured and controlled manner, handle runtime errors. A principal advantage of exception handling is that it automates much of the error-handling code that previously had to be entered "by hand" into any large program. For example, in a computer language without exception handling, error codes must be returned when a method fails, and these values must be checked manually each time the method is called. This approach is both tedious and error-prone. Exception handling streamlines error-handling by allowing your program to define a block of code, called an *exception handler,* that is executed automatically when an error occurs. It is not necessary to manually check the success or failure of each specific operation or method call. If an error occurs, it will be processed by the exception handler.

Exception handling is also important because C# defines standard exceptions for common program errors, such as divide-by-zero or index-out-of-range. To respond to these errors, your program must watch for and handle these exceptions. In the final analysis, to be a successful C# programmer means that you are fully capable of navigating C#'s exception-handling subsystem.

The System.Exception Class

In C#, exceptions are represented by classes. All exception classes must be derived from the built-in exception class **Exception**, which is part of the **System** namespace. Thus, all exceptions are subclasses of **Exception**.

One very important subclass of **Exception** is **SystemException**. This is the exception class from which all exceptions generated by the C# runtime system (that is, the CLR) are derived. **SystemException** does not add anything to **Exception**. It simply defines the top of the standard exceptions hierarchy.

The .NET Framework defines several built-in exceptions that are derived from **SystemException**. For example, when a division-by-zero is attempted, a **DivideByZeroException** exception is generated. As you will see later in this chapter, you can create your own exception classes by deriving them from **Exception**.

Exception Handling Fundamentals

C# exception handling is managed via four keywords: **try**, **catch**, **throw**, and **finally**. They form an interrelated subsystem in which the use of one implies the use of another. Throughout the course of this chapter, each keyword is examined in detail. However, it is useful at the outset to have a general understanding of the role each plays in exception handling. Briefly, here is how they work.

Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is *thrown.* Your code can catch this exception using **catch** and handle it in some rational manner. System-generated exceptions are automatically thrown by the runtime system. To manually throw an exception, use the keyword **throw**. Any code that absolutely must be executed upon exiting from a **try** block is put in a **finally** block.

Using try and catch

At the core of exception handling are **try** and **catch**. These keywords work together, and you can't have a **catch** without a **try**. Here is the general form of the **try**/**catch** exception-handling blocks:

```
try {
  // block of code to monitor for errors
}

catch (ExcepType1 exOb) {
  // handler for ExcepType1
}

catch (ExcepType2 exOb) {
  // handler for ExcepType2
}
.
.
.
```

Here, *ExcepType* is the type of exception that has occurred. When an exception is thrown, it is caught by its corresponding **catch** clause, which then processes the exception. As the general form shows, more than one **catch** clause can be associated with a **try**. The type of the exception determines which **catch** is executed. That is, if the exception type specified by a **catch** matches that of the exception, then that **catch** is executed (and all others are bypassed). When an exception is caught, the exception variable *exOb* will receive its value.

Actually, specifying *exOb* is optional. If the exception handler does not need access to the exception object (as is often the case), there is no need to specify *exOb.* The exception type alone is sufficient. For this reason, many of the examples in this chapter will not specify *exOb.*

Here is an important point: If no exception is thrown, then a **try** block ends normally, and all of its **catch** clauses are bypassed. Execution resumes with the first statement following the last **catch**. Thus, a **catch** is executed only if an exception is thrown.

A Simple Exception Example

Here is a simple example that illustrates how to watch for and catch an exception. As you know, it is an error to attempt to index an array beyond its boundaries. When this error occurs, the CLR throws an **IndexOutOfRangeException**, which is a standard exception

defined by the .NET Framework. The following program purposely generates such an exception and then catches it:

```
// Demonstrate exception handling.

using System;

class ExcDemo1 {
  static void Main() {
    int[] nums = new int[4];

    try {
      Console.WriteLine("Before exception is generated.");

      // Generate an index out-of-bounds exception.
      for(int i=0; i < 10; i++) {
        nums[i] = i;
        Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
      }

      Console.WriteLine("this won't be displayed");
    }
    catch (IndexOutOfRangeException) {
      // Catch the exception.
      Console.WriteLine("Index out-of-bounds!");
    }
    Console.WriteLine("After catch block.");
  }
}
```

This program displays the following output:

```
Before exception is generated.
nums[0]: 0
nums[1]: 1
nums[2]: 2
nums[3]: 3
Index out-of-bounds!
After catch block.
```

Notice that **nums** is an **int** array of four elements. However, the **for** loop tries to index **nums** from 0 to 9, which causes an **IndexOutOfRangeException** to occur when an index value of 4 is tried.

Although quite short, the preceding program illustrates several key points about exception handling. First, the code that you want to monitor for errors is contained within a **try** block. Second, when an exception occurs (in this case, because of the attempt to index **nums** beyond its bounds inside the **for** loop), the exception is thrown out of the **try** block and caught by the **catch**. At this point, control passes to the **catch** block, and the **try** block is terminated. That is, **catch** is *not* called. Rather, program execution is transferred to it. Thus, the **WriteLine( )** statement following the out-of-bounds index will never execute. After the **catch** block executes, program control continues with the statements following the **catch**. Thus, it is the job of your exception handler to remedy the problem that caused the exception so program execution can continue normally.

Notice that no exception variable is specified in the **catch** clause. Instead, only the type of the exception (**IndexOutOfRangeException** in this case) is required. As mentioned, an exception variable is needed only when access to the exception object is required. In some cases, the value of the exception object can be used by the exception handler to obtain additional information about the error, but in many cases, it is sufficient to simply know that an exception occurred. Thus, it is not unusual for the **catch** variable to be absent in the exception handler, as is the case in the preceding program.

As explained, if no exception is thrown by a **try** block, no **catch** will be executed and program control resumes after the **catch**. To confirm this, in the preceding program, change the **for** loop from

```
for(int i=0; i < 10; i++) {
```

to

```
for(int i=0; i < nums.Length; i++) {
```

Now, the loop does not overrun **nums**' boundary. Thus, no exception is generated, and the **catch** block is not executed.

A Second Exception Example

It is important to understand that all code executed within a **try** block is monitored for exceptions. This includes exceptions that might be generated by a method called from within the **try** block. An exception thrown by a method called from within a **try** block can be caught by that **try** block, assuming, of course, that the method itself did not catch the exception.

For example, consider the following program. **Main()** establishes a **try** block from which the method **GenException()** is called. Inside **GenException()**, an **IndexOutOfRangeException** is generated. This exception is not caught by **GenException()**. However, since **GenException()** was called from within a **try** block in **Main()**, the exception is caught by the **catch** statement associated with that **try**.

```
/* An exception can be generated by one
   method and caught by another. */

using System;

class ExcTest {
  // Generate an exception.
  public static void GenException() {
    int[] nums = new int[4];

    Console.WriteLine("Before exception is generated.");

    // Generate an index out-of-bounds exception.
    for(int i=0; i < 10; i++) {
      nums[i] = i;
      Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
    }
```

```
    Console.WriteLine("this won't be displayed");
  }
}

class ExcDemo2 {
  static void Main() {

    try {
      ExcTest.GenException();
    }
    catch (IndexOutOfRangeException) {
      // Catch the exception.
      Console.WriteLine("Index out-of-bounds!");
    }
    Console.WriteLine("After catch block.");
  }
}
```

This program produces the following output, which is the same as that produced by the first version of the program shown earlier:

```
Before exception is generated.
nums[0]: 0
nums[1]: 1
nums[2]: 2
nums[3]: 3
Index out-of-bounds!
After catch block.
```

As explained, because **GenException( )** is called from within a **try** block, the exception that it generates (and does not catch) is caught by the **catch** in **Main( )**. Understand, however, that if **GenException( )** had caught the exception, then it never would have been passed back to **Main( )**.

The Consequences of an Uncaught Exception

Catching one of the standard exceptions, as the preceding program does, has a side benefit: It prevents abnormal program termination. When an exception is thrown, it must be caught by some piece of code, somewhere. In general, if your program does not catch an exception, it will be caught by the runtime system. The trouble is that the runtime system will report an error and terminate the program. For instance, in this example, the index out-of-bounds exception is not caught by the program:

```
// Let the C# runtime system handle the error.

using System;

class NotHandled {
  static void Main() {
    int[] nums = new int[4];
```

```
    Console.WriteLine("Before exception is generated.");

    // Generate an index out-of-bounds exception.
    for(int i=0; i < 10; i++) {
      nums[i] = i;
      Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
    }

  }
}
```

When the array index error occurs, execution is halted and the following error message is displayed:

```
Unhandled Exception: System.IndexOutOfRangeException:
          Index was outside the bounds of the array.
   at NotHandled.Main()
```

Although such a message is useful while debugging, you would not want others to see it, to say the least! This is why it is important for your program to handle exceptions itself.

As mentioned earlier, the type of the exception must match the type specified in a **catch**. If it doesn't, the exception won't be caught. For example, the following program tries to catch an array boundary error with a **catch** for a **DivideByZeroException** (another built-in exception). When the array boundary is overrun, an **IndexOutOfRangeException** is generated, but it won't be caught by the **catch**. This results in abnormal program termination.

```
// This won't work!

using System;

class ExcTypeMismatch {
  static void Main() {
    int[] nums = new int[4];

    try {
      Console.WriteLine("Before exception is generated.");

      // Generate an index out-of-bounds exception.
      for(int i=0; i < 10; i++) {
        nums[i] = i;
        Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
      }

      Console.WriteLine("this won't be displayed");
    }

    /* Can't catch an array boundary error with a
       DivideByZeroException. */
    catch (DivideByZeroException) {
      // Catch the exception.
      Console.WriteLine("Index out-of-bounds!");
    }
```

```
    Console.WriteLine("After catch block.");
  }
}
```

The output is shown here:

```
Before exception is generated.
nums[0]: 0
nums[1]: 1
nums[2]: 2
nums[3]: 3

Unhandled Exception: System.IndexOutOfRangeException:
          Index was outside the bounds of the array.
   at ExcTypeMismatch.Main()
```

As the output demonstrates, a **catch** for **DivideByZeroException** won't catch an **IndexOutOfRangeException**.

Exceptions Let You Handle Errors Gracefully

One of the key benefits of exception handling is that it enables your program to respond to an error and then continue running. For example, consider the following example that divides the elements of one array by the elements of another. If a division-by-zero occurs, a **DivideByZeroException** is generated. In the program, this exception is handled by reporting the error and then continuing with execution. Thus, attempting to divide by zero does not cause an abrupt runtime error resulting in the termination of the program. Instead, it is handled gracefully, allowing program execution to continue.

```
// Handle error gracefully and continue.

using System;

class ExcDemo3 {
  static void Main() {
    int[] numer = { 4, 8, 16, 32, 64, 128 };
    int[] denom = { 2, 0, 4, 4, 0, 8 };

    for(int i=0; i < numer.Length; i++) {
      try {
        Console.WriteLine(numer[i] + " / " +
                          denom[i] + " is " +
                          numer[i]/denom[i]);
      }
      catch (DivideByZeroException) {
        // Catch the exception.
        Console.WriteLine("Can't divide by Zero!");
      }
    }
  }
}
```

The output from the program is shown here:

```
4 / 2 is 2
Can't divide by Zero!
16 / 4 is 4
32 / 4 is 8
Can't divide by Zero!
128 / 8 is 16
```

This example makes another important point: Once an exception has been handled, it is removed from the system. Therefore, in the program, each pass through the loop enters the **try** block anew—any prior exceptions have been handled. This enables your program to handle repeated errors.

Using Multiple catch Clauses

You can associate more than one **catch** clause with a **try**. In fact, it is common to do so. However, each **catch** must catch a different type of exception. For example, the program shown here catches both array boundary and divide-by-zero errors:

```
// Use multiple catch clauses.

using System;

class ExcDemo4 {
  static void Main() {
    // Here, numer is longer than denom.
    int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int[] denom = { 2, 0, 4, 4, 0, 8 };

    for(int i=0; i < numer.Length; i++) {
      try {
        Console.WriteLine(numer[i] + " / " +
                          denom[i] + " is " +
                          numer[i]/denom[i]);
      }
      catch (DivideByZeroException) {
        Console.WriteLine("Can't divide by Zero!");
      }
      catch (IndexOutOfRangeException) {
        Console.WriteLine("No matching element found.");
      }
    }
  }
}
```

This program produces the following output:

```
4 / 2 is 2
Can't divide by Zero!
16 / 4 is 4
32 / 4 is 8
Can't divide by Zero!
128 / 8 is 16
```

```
No matching element found.
No matching element found.
```

As the output confirms, each **catch** clause responds only to its own type of exception.

In general, **catch** clauses are checked in the order in which they occur in a program. Only the first matching clause is executed. All other **catch** blocks are ignored.

Catching All Exceptions

Occasionally, you might want to catch all exceptions, no matter the type. To do this, use a **catch** clause that specifies no exception type or variable. It has this general form:

```
catch {
  // handle exceptions
}
```

This creates a "catch all" handler that ensures that all exceptions are caught by your program.

Here is an example of a "catch all" exception handler. Notice that it catches both the **IndexOutOfRangeException** and the **DivideByZeroException** generated by the program:

```
// Use the "catch all" catch.

using System;

class ExcDemo5 {
  static void Main() {
    // Here, numer is longer than denom.
    int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int[] denom = { 2, 0, 4, 4, 0, 8 };

    for(int i=0; i < numer.Length; i++) {
      try {
        Console.WriteLine(numer[i] + " / " +
                          denom[i] + " is " +
                          numer[i]/denom[i]);
      }
      catch { // A "catch-all" catch.
        Console.WriteLine("Some exception occurred.");
      }
    }
  }
}
```

The output is shown here:

```
4 / 2 is 2
Some exception occurred.
16 / 4 is 4
32 / 4 is 8
Some exception occurred.
128 / 8 is 16
Some exception occurred.
Some exception occurred.
```

There is one point to remember about using a catch-all **catch**: It must be the last **catch** clause in the **catch** sequence.

NOTE *In the vast majority of cases you should not use the "catch all" handler as a means of dealing with exceptions. It is normally better to deal individually with the exceptions that your code can generate. The inappropriate use of the "catch all" handler can lead to situations in which errors that would otherwise be caught during testing are masked. It is also difficult to correctly handle all types of exceptions with a single hander. That said, a "catch all" handler might be appropriate in certain specialized circumstances, such as in a runtime code analysis tool.*

Nesting try Blocks

One **try** block can be nested within another. An exception generated within the inner **try** block that is not caught by a **catch** associated with that **try** is propagated to the outer **try** block. For example, here the **IndexOutOfRangeException** is not caught by the inner **try** block, but by the outer **try**:

```
// Use a nested try block.

using System;

class NestTrys {
  static void Main() {
    // Here, numer is longer than denom.
    int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int[] denom = { 2, 0, 4, 4, 0, 8 };

    try { // outer try
      for(int i=0; i < numer.Length; i++) {
        try { // nested try
          Console.WriteLine(numer[i] + " / " +
                            denom[i] + " is " +
                            numer[i]/denom[i]);
        }
        catch (DivideByZeroException) {
          Console.WriteLine("Can't divide by Zero!");
        }
      }
    }
    catch (IndexOutOfRangeException) {
      Console.WriteLine("No matching element found.");
      Console.WriteLine("Fatal error -- program terminated.");
    }
  }
}
```

The output from the program is shown here:

```
4 / 2 is 2
Can't divide by Zero!
16 / 4 is 4
```

```
32 / 4 is 8
Can't divide by Zero!
128 / 8 is 16
No matching element found.
Fatal error -- program terminated.
```

In this example, an exception that can be handled by the inner **try**—in this case a divide-by-zero error—allows the program to continue. However, an array boundary error is caught by the outer **try**, which causes the program to terminate.

Although certainly not the only reason for nested **try** statements, the preceding program makes an important point that can be generalized. Often, nested **try** blocks are used to allow different categories of errors to be handled in different ways. Some types of errors are catastrophic and cannot be fixed. Some are minor and can be handled immediately. Many programmers use an outer **try** block to catch the most severe errors, allowing inner **try** blocks to handle less serious ones. You can also use an outer **try** block as a "catch all" block for those errors that are not handled by the inner block.

Throwing an Exception

The preceding examples have been catching exceptions generated automatically by the runtime system. However, it is possible to throw an exception manually by using the **throw** statement. Its general form is shown here:

throw *exceptOb*;

The *exceptOb* must be an object of an exception class derived from **Exception**.

Here is an example that illustrates the **throw** statement by manually throwing a **DivideByZeroException**:

```
// Manually throw an exception.

using System;

class ThrowDemo {
  static void Main() {
    try {
      Console.WriteLine("Before throw.");
      throw new DivideByZeroException();
    }
    catch (DivideByZeroException) {
      Console.WriteLine("Exception caught.");
    }
    Console.WriteLine("After try/catch statement.");
  }
}
```

The output from the program is shown here:

```
Before throw.
Exception caught.
After try/catch statement.
```

Notice how the **DivideByZeroException** was created using **new** in the **throw** statement. Remember, **throw** throws an object. Thus, you must create an object for it to throw. That is, you can't just throw a type. In this case, the default constructor is used to create a **DivideByZeroException** object, but other constructors are available for exceptions.

Most often, exceptions that you throw will be instances of exception classes that you created. As you will see later in this chapter, creating your own exception classes allows you to handle errors in your code as part of your program's overall exception handling strategy.

Rethrowing an Exception

An exception caught by one **catch** can be rethrown so that it can be caught by an outer **catch**. The most likely reason for rethrowing an exception is to allow multiple handlers access to the exception. For example, perhaps one exception handler manages one aspect of an exception, and a second handler copes with another aspect. To rethrow an exception, you simply specify **throw**, without specifying an expression. That is, you use this form of **throw**:

```
throw ;
```

Remember, when you rethrow an exception, it will not be recaught by the same **catch** clause. Instead, it will propagate to an outer **catch**.

The following program illustrates rethrowing an exception. In this case, it rethrows an **IndexOutOfRangeException**.

```
// Rethrow an exception.

using System;

class Rethrow {
  public static void GenException() {
    // Here, numer is longer than denom.
    int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int[] denom = { 2, 0, 4, 4, 0, 8 };

    for(int i=0; i<numer.Length; i++) {
      try {
        Console.WriteLine(numer[i] + " / " +
                          denom[i] + " is " +
                          numer[i]/denom[i]);
      }
      catch (DivideByZeroException) {
        Console.WriteLine("Can't divide by Zero!");
      }
      catch (IndexOutOfRangeException) {
        Console.WriteLine("No matching element found.");
        throw; // rethrow the exception
      }
    }
  }
}

class RethrowDemo {
  static void Main() {
    try {
```

```
      Rethrow.GenException();
    }
    catch(IndexOutOfRangeException) {
      // recatch exception
     Console.WriteLine("Fatal error -- " + "program terminated.");
    }
  }
}
```

In this program, divide-by-zero errors are handled locally, by **GenException( )**, but an array boundary error is rethrown. In this case, the **IndexOutOfRangeException** is handled by **Main( )**.

Using finally

Sometimes you will want to define a block of code that will execute when a **try/catch** block is left. For example, an exception might cause an error that terminates the current method, causing its premature return. However, that method may have opened a file or a network connection that needs to be closed. Such types of circumstances are common in programming, and C# provides a convenient way to handle them: **finally**.

To specify a block of code to execute when a **try/catch** block is exited, include a **finally** block at the end of a **try/catch** sequence. The general form of a **try/catch** that includes **finally** is shown here:

```
try {
  // block of code to monitor for errors
}

catch (ExcepType1 exOb) {
  // handler for ExcepType1
}

catch (ExcepType2 exOb) {
  // handler for ExcepType2
}
.
.
.
finally {
  // finally code
}
```

The **finally** block will be executed whenever execution leaves a **try/catch** block, no matter what conditions cause it. That is, whether the **try** block ends normally, or because of an exception, the last code executed is that defined by **finally**. The **finally** block is also executed if any code within the **try** block or any of its **catch** blocks returns from the method.

Here is an example of **finally**:

```
// Use finally.

using System;

class UseFinally {
  public static void GenException(int what) {
```

```
    int t;
    int[] nums = new int[2];

    Console.WriteLine("Receiving " + what);
    try {
      switch(what) {
        case 0:
          t = 10 / what; // generate div-by-zero error
          break;
        case 1:
          nums[4] = 4; // generate array index error
          break;
        case 2:
          return; // return from try block
      }
    }
    catch (DivideByZeroException) {
      Console.WriteLine("Can't divide by Zero!");
      return; // return from catch
    }
    catch (IndexOutOfRangeException) {
      Console.WriteLine("No matching element found.");
    }
    finally {
      Console.WriteLine("Leaving try.");
    }
  }
}

class FinallyDemo {
  static void Main() {

    for(int i=0; i < 3; i++) {
      UseFinally.GenException(i);
      Console.WriteLine();
    }
  }
}
```

Here is the output produced by the program:

```
Receiving 0
Can't divide by Zero!
Leaving try.

Receiving 1
No matching element found.
Leaving try.

Receiving 2
Leaving try.
```

As the output shows, no matter how the **try** block is exited, the **finally** block executed.

One other point: Syntactically, when a **finally** block follows a **try** block, no **catch** clauses are technically required. Thus, you can have a **try** followed by a **finally** with no **catch** clauses. In this case, the **finally** block is executed when the **try** exits, but no exceptions are handled.

A Closer Look at the Exception Class

Up to this point, we have been catching exceptions, but we haven't been doing anything with the exception object itself. As explained earlier, a **catch** clause allows you to specify an exception type *and* a variable. The variable receives a reference to the exception object. Since all exceptions are derived from **Exception**, all exceptions support the members defined by **Exception**. Here we will examine several of its most useful members and constructors, and put the exception variable to use.

Exception defines several properties. Three of the most interesting are **Message**, **StackTrace**, and **TargetSite**. All are read-only. **Message** contains a string that describes the nature of the error. **StackTrace** contains a string that contains the stack of calls that lead to the exception. **TargetSite** obtains an object that specifies the method that generated the exception.

Exception also defines several methods. One that you will often use is **ToString( )**, which returns a string that describes the exception. **ToString( )** is automatically called when an exception is displayed via **WriteLine( )**, for example.

The following program demonstrates these properties and this method:

```
// Using Exception members.

using System;

class ExcTest {
  public static void GenException() {
    int[] nums = new int[4];

    Console.WriteLine("Before exception is generated.");

    // Generate an index out-of-bounds exception.
      for(int i=0; i < 10; i++) {
        nums[i] = i;
        Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
      }

    Console.WriteLine("this won't be displayed");
  }
}

class UseExcept {
  static void Main() {

    try {
      ExcTest.GenException();
    }
    catch (IndexOutOfRangeException exc) {
```

```
      Console.WriteLine("Standard message is: ");
      Console.WriteLine(exc); // calls ToString()
      Console.WriteLine("Stack trace: " + exc.StackTrace);
      Console.WriteLine("Message: " + exc.Message);
      Console.WriteLine("TargetSite: " + exc.TargetSite);
    }
    Console.WriteLine("After catch block.");
  }
}
```

The output from this program is shown here:

```
Before exception is generated.
nums[0]: 0
nums[1]: 1
nums[2]: 2
nums[3]: 3
Standard message is:
System.IndexOutOfRangeException: Index was outside the bounds of the array.
   at ExcTest.GenException()
   at UseExcept.Main()
Stack trace:    at ExcTest.GenException()
   at UseExcept.Main()
Message: Index was outside the bounds of the array.
TargetSite: Void GenException()
After catch block.
```

Exception defines the following four constructors:

public Exception()

public Exception(string *str*)

public Exception(string *str*, Exception *inner*)

protected Exception(System.Runtime.Serialization.SerializationInfo *si*,
System.Runtime.Serialization.StreamingContext *sc*)

The first is the default constructor. The second specifies the string associated with the **Message** property associated with the exception. The third specifies what is called an *inner exception*. It is used when one exception gives rise to another. In this case, *inner* specifies the first exception, which will be null if no inner exception exists. (The inner exception, if it exists, can be obtained from the **InnerException** property defined by **Exception**.) The last constructor handles exceptions that occur remotely and require deserialization.

One other point: In the fourth **Exception** constructor shown above, notice that the types **SerializationInfo** and **StreamingContext** are contained in the **System.Runtime.Serialization** namespace.

Commonly Used Exceptions

The **System** namespace defines several standard, built-in exceptions. All are derived from **SystemException** since they are generated by the CLR when runtime errors occur. Several of the more commonly used standard exceptions are shown in Table 13-1.

Exception	Meaning
ArrayTypeMismatchException	Type of value being stored is incompatible with the type of the array.
DivideByZeroException	Division by zero attempted.
IndexOutOfRangeException	Array index is out of bounds.
InvalidCastException	A runtime cast is invalid.
OutOfMemoryException	Insufficient free memory exists to continue program execution. For example, this exception will be thrown if there is not sufficient free memory to create an object via **new**.
OverflowException	An arithmetic overflow occurred.
NullReferenceException	An attempt was made to operate on a null reference—that is, a reference that does not refer to an object.
StackOverflowException	The stack was overrun.

TABLE 13-1 Commonly Used Exceptions Defined Within the **System** Namespace

Most of the exceptions in Table 13-1 are self-explanatory, with the possible exception of **NullReferenceException**. This exception is thrown when there is an attempt to use a null reference as if it referred to an object—for example, if you attempt to call a method on a null reference. A *null reference* is a reference that does not point to any object. One way to create a null reference is to explicitly assign it the value null by using the keyword **null**. Null references can also occur in other ways that are less obvious. Here is a program that demonstrates the **NullReferenceException**:

```
// Use the NullReferenceException.

using System;

class X {
  int x;
  public X(int a) {
    x = a;
  }

  public int Add(X o) {
    return x + o.x;
  }
}

// Demonstrate NullReferenceException.
class NREDemo {
  static void Main() {
    X p = new X(10);
    X q = null; // q is explicitly assigned null
    int val;
```

```
    try {
      val = p.Add(q); // this will lead to an exception
    } catch (NullReferenceException) {
      Console.WriteLine("NullReferenceException!");
      Console.WriteLine("fixing...\n");

      // Now, fix it.
      q = new X(9);
      val = p.Add(q);
    }

    Console.WriteLine("val is {0}", val);
  }
}
```

The output from the program is shown here:

```
NullReferenceException!
fixing...

val is 19
```

The program creates a class called **X** that defines a member called **x** and the **Add( )** method, which adds the invoking object's **x** to the **x** in the object passed as a parameter. In **Main( )**, two **X** objects are created. The first, **p**, is initialized. The second, **q**, is not. Instead, it is explicitly assigned **null**. Then **p.Add( )** is called with **q** as an argument. Because **q** does not refer to any object, a **NullReferenceException** is generated when the attempt is made to obtain the value of **q.x**.

An interesting exception is **StackOverflowException**, which is thrown when the system stack is overrun. One situation in which this can happen is when a recursive method runs wild. Because the stack is exhausted, a **StackOverflowException** can't be caught by your program. Instead, a stack overflow results in the abnormal termination of your program.

Deriving Exception Classes

Although C#'s built-in exceptions handle most common errors, C#'s exception handling mechanism is not limited to these errors. In fact, part of the power of C#'s approach to exceptions is its ability to handle exceptions that you create. You can use custom exceptions to handle errors in your own code. Creating an exception is easy. Just define a class derived from **Exception**. Your derived classes don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.

NOTE *In the past, custom exceptions were derived from* ***ApplicationException*** *since this is the hierarchy that was originally reserved for application-related exceptions. However, Microsoft no longer recommends this. Instead, at the time of this writing, Microsoft recommends deriving custom exceptions from* ***Exception****. For this reason, this approach is used here.*

The exception classes that you create will automatically have the properties and methods defined by **Exception** available to them. Of course, you can override one or more of these members in exception classes that you create.

When creating your own exception class, you will generally want your class to support all of the constructors defined by **Exception**. For simple custom exception classes, this is easy to do because you can simply pass along the constructor's arguments to the corresponding **Exception** constructor via **base**. Of course, technically, you need to provide only those constructors actually used by your program.

Here is an example that makes use of a custom exception type. At the end of Chapter 10 an array class called **RangeArray** was developed. As you may recall, **RangeArray** supports single-dimensional **int** arrays in which the starting and ending index is specified by the user. For example, an array that ranges from –5 to 27 is perfectly legal for a **RangeArray**. In Chapter 10, if an index was out of range, a special error variable defined by **RangeArray** was set. This meant that the error variable had to be checked after each operation by the code that used **RangeArray**. Of course, such an approach is error-prone and clumsy. A far better design is to have **RangeArray** throw a custom exception when a range error occurs. This is precisely what the following version of **RangeArray** does:

```
// Use a custom Exception for RangeArray errors.

using System;

// Create a RangeArray exception.
class RangeArrayException : Exception {
  /* Implement all of the Exception constructors. Notice that
     the constructors simply execute the base class constructor.
     Because RangeArrayException adds nothing to Exception,
     there is no need for any further actions. */
  public RangeArrayException() : base() { }
  public RangeArrayException(string str) : base(str) { }
  public RangeArrayException(string str, Exception inner) :
    base(str, inner) { }
  protected RangeArrayException(
    System.Runtime.Serialization.SerializationInfo si,
    System.Runtime.Serialization.StreamingContext sc) :
      base(si, sc) { }

  // Override ToString for RangeArrayException.
  public override string ToString() {
    return Message;
  }
}

// An improved version of RangeArray.
class RangeArray {
  // Private data.
  int[] a; // reference to underlying array
  int lowerBound; // smallest index
  int upperBound; // largest index

  // An auto-implemented, read-only Length property.
  public int Length { get; private set; }
```

```
  // Construct array given its size.
  public RangeArray(int low, int high) {
    high++;
    if(high <= low) {
      throw new RangeArrayException("Low index not less than high.");
    }
    a = new int[high - low];
    Length = high - low;

    lowerBound = low;
    upperBound = --high;
  }

  // This is the indexer for RangeArray.
  public int this[int index] {
    // This is the get accessor.
    get {
      if(ok(index)) {
        return a[index - lowerBound];
      } else {
        throw new RangeArrayException("Range Error.");
      }
    }

    // This is the set accessor.
    set {
      if(ok(index)) {
        a[index - lowerBound] = value;
      }
      else throw new RangeArrayException("Range Error.");
    }
  }

  // Return true if index is within bounds.
  private bool ok(int index) {
    if(index >= lowerBound & index <= upperBound) return true;
    return false;
  }
}

// Demonstrate the index-range array.
class RangeArrayDemo {
  static void Main() {
    try {
      RangeArray ra = new RangeArray(-5, 5);
      RangeArray ra2 = new RangeArray(1, 10);

      // Demonstrate ra.
      Console.WriteLine("Length of ra: " + ra.Length);

      for(int i = -5; i <= 5; i++)
        ra[i] = i;
```

```
      Console.Write("Contents of ra: ");
      for(int i = -5; i <= 5; i++)
        Console.Write(ra[i] + " ");

      Console.WriteLine("\n");

      // Demonstrate ra2.
      Console.WriteLine("Length of ra2: " + ra2.Length);

      for(int i = 1; i <= 10; i++)
        ra2[i] = i;

      Console.Write("Contents of ra2: ");
      for(int i = 1; i <= 10; i++)
        Console.Write(ra2[i] + " ");

      Console.WriteLine("\n");

    } catch (RangeArrayException exc) {
       Console.WriteLine(exc);
    }

    // Now, demonstrate some errors.
    Console.WriteLine("Now generate some range errors.");

    // Use an invalid constructor.
    try {
      RangeArray ra3 = new RangeArray(100, -10); // Error
    } catch (RangeArrayException exc) {
       Console.WriteLine(exc);
    }

    // Use an invalid index.
    try {
      RangeArray ra3 = new RangeArray(-2, 2);

      for(int i = -2; i <= 2; i++)
        ra3[i] = i;

      Console.Write("Contents of ra3: ");
      for(int i = -2; i <= 10; i++) // generate range error
        Console.Write(ra3[i] + " ");

    } catch (RangeArrayException exc) {
       Console.WriteLine(exc);
    }
  }
}
```

The output from the program is shown here:

```
Length of ra: 11
Contents of ra: -5 -4 -3 -2 -1 0 1 2 3 4 5
```

```
Length of ra2: 10
Contents of ra2: 1 2 3 4 5 6 7 8 9 10

Now generate some range errors.
Low index not less than high.
Contents of ra3: -2 -1 0 1 2 Range Error.
```

When a range error occurs, **RangeArray** throws an object of type **RangeArrayException**. Notice there are three places in **RangeArray** that this might occur: in the **get** indexer accessor, in the **set** indexer accessor, and by the **RangeArray** constructor. To catch these exceptions implies that **RangeArray** objects must be constructed and accessed from within a **try** block, as the program illustrates. By using an exception to report errors, **RangeArray** now acts like one of C#'s built-in types and can be fully integrated into a program's exception-handling mechanism.

Notice that none of the **RangeArrayException** constructors provide any statements in their body. Instead, they simply pass their arguments along to **Exception** via **base**. As explained, in cases in which your exception class does not add any functionality, you can simply let the **Exception** constructors handle the process. There is no requirement that your derived class add anything to what is inherited from **Exception**.

Before moving on, you might want to experiment with this program a bit. For example, try commenting-out the override of **ToString( )** and observe the results. Also, try creating an exception using the default constructor, and observe what C# generates as its default message.

Catching Derived Class Exceptions

You need to be careful how you order **catch** clauses when trying to catch exception types that involve base and derived classes, because a **catch** for a base class will also match any of its derived classes. For example, because the base class of all exceptions is **Exception**, catching **Exception** catches all possible exceptions. Of course, using **catch** without an exception type provides a cleaner way to catch all exceptions, as described earlier. However, the issue of catching derived class exceptions is very important in other contexts, especially when you create exceptions of your own.

If you want to catch exceptions of both a base class type and a derived class type, put the derived class first in the **catch** sequence. This is necessary because a base class **catch** will also catch all derived classes. Fortunately, this rule is self-enforcing because putting the base class first causes a compile-time error.

The following program creates two exception classes called **ExceptA** and **ExceptB**. **ExceptA** is derived from **Exception**. **ExceptB** is derived from **ExceptA**. The program then throws an exception of each type. For brevity, the custom exceptions supply only one constructor (which takes a string that describes the exception). But remember, in commercial code, your custom exception classes will normally provide all four of the constructors defined by **Exception**.

```
// Derived exceptions must appear before base class exceptions.

using System;

// Create an exception.
class ExceptA : Exception {
  public ExceptA(string str) : base(str) { }
```

```
  public override string ToString() {
    return Message;
  }
}

// Create an exception derived from ExceptA.
class ExceptB : ExceptA {
  public ExceptB(string str) : base(str) { }

  public override string ToString() {
    return Message;
  }
}

class OrderMatters {
  static void Main() {
    for(int x = 0; x < 3; x++) {
      try {
        if(x==0) throw new ExceptA("Caught an ExceptA exception");
        else if(x==1) throw new ExceptB("Caught an ExceptB exception");
        else throw new Exception();
      }
      catch (ExceptB exc) {
        Console.WriteLine(exc);
      }
      catch (ExceptA exc) {
        Console.WriteLine(exc);
      }
      catch (Exception exc) {
        Console.WriteLine(exc);
      }
    }
  }
}
```

The output from the program is shown here:

```
Caught an ExceptA exception
Caught an ExceptB exception
System.Exception: Exception of type 'System.Exception' was thrown.
   at OrderMatters.Main()
```

Notice the type and order of the **catch** clauses. This is the only order in which they can occur. Since **ExceptB** is derived from **ExceptA**, the **catch** for **ExceptB** must be before the one for **ExceptA**. Similarly, the **catch** for **Exception** (which is the base class for all exceptions) must appear last. To prove this point for yourself, try rearranging the **catch** clauses. Doing so will result in a compile-time error.

One good use of a base class **catch** clause is to catch an entire category of exceptions. For example, imagine you are creating a set of exceptions for some device. If you derive all of the exceptions from a common base class, then applications that don't need to know precisely what problem occurred could simply catch the base class exception, avoiding the unnecessary duplication of code.

Using checked and unchecked

A special feature in C# relates to the generation of overflow exceptions in arithmetic computations. As you know, it is possible for some types of arithmetic computations to produce a result that exceeds the range of the data type involved in the computation. When this occurs, the result is said to *overflow.* For example, consider the following sequence:

```
byte a, b, result;
a = 127;
b = 127;

result = (byte)(a * b);
```

Here, the product of **a** and **b** exceeds the range of a **byte** value. Thus, the result overflows the type of the result.

C# allows you to specify whether your code will raise an exception when overflow occurs by using the keywords **checked** and **unchecked**. To specify that an expression be checked for overflow, use **checked**. To specify that overflow be ignored, use **unchecked**. In this case, the result is truncated to fit into the target type of the expression.

The **checked** keyword has these two general forms. One checks a specific expression and is called the *operator form* of **checked**. The other checks a block of statements and is called the *statement form.*

```
checked (expr)

checked {
  // statements to be checked
}
```

Here, *expr* is the expression being checked. If a checked expression overflows, then an **OverflowException** is thrown.

The **unchecked** keyword also has two general forms. The first is the operator form, which ignores overflow for a specific expression. The second ignores overflow for a block of statements.

```
unchecked (expr)

unchecked {
  // statements for which overflow is ignored
}
```

Here, *expr* is the expression that is not being checked for overflow. If an unchecked expression overflows, then truncation will occur.

Here is a program that demonstrates both **checked** and **unchecked**:

```
// Using checked and unchecked.

using System;

class CheckedDemo {
  static void Main() {
    byte a, b;
```

```
    byte result;

    a = 127;
    b = 127;

    try {
      result = unchecked((byte)(a * b));
      Console.WriteLine("Unchecked result: " + result);

      result = checked((byte)(a * b)); // this causes exception
      Console.WriteLine("Checked result: " + result); // won't execute
    }
    catch (OverflowException exc) {
      Console.WriteLine(exc);
    }
  }
}
```

The output from the program is shown here:

```
Unchecked result: 1
System.OverflowException: Arithmetic operation resulted in an overflow.
   at CheckedDemo.Main()
```

As is evident, the unchecked expression resulted in a truncation. The checked expression caused an exception.

The preceding program demonstrated the use of **checked** and **unchecked** for a single expression. The following program shows how to check and uncheck a block of statements.

```
// Using checked and unchecked with statement blocks.

using System;

class CheckedBlocks {
  static void Main() {
    byte a, b;
    byte result;

    a = 127;
    b = 127;

    try {
      unchecked {
        a = 127;
        b = 127;
        result = unchecked((byte)(a * b));
        Console.WriteLine("Unchecked result: " + result);

        a = 125;
        b = 5;
        result = unchecked((byte)(a * b));
        Console.WriteLine("Unchecked result: " + result);
      }
```

```
      checked {
        a = 2;
        b = 7;
        result = checked((byte)(a * b)); // this is OK
        Console.WriteLine("Checked result: " + result);

        a = 127;
        b = 127;
        result = checked((byte)(a * b)); // this causes exception
        Console.WriteLine("Checked result: " + result); // won't execute
      }
    }
    catch (OverflowException exc) {
      Console.WriteLine(exc);
    }
  }
}
```

The output from the program is shown here:

```
Unchecked result: 1
Unchecked result: 113
Checked result: 14
System.OverflowException: Arithmetic operation resulted in an overflow.
   at CheckedBlocks.Main()
```

As you can see, the unchecked block results in the overflow being truncated. When overflow occurred in the checked block, an exception was raised.

One reason that you may need to use **checked** or **unchecked** is that the default checked/unchecked status of overflow is determined by the setting of a compiler option and the execution environment, itself. Thus, for some types of programs, it is best to specify the overflow check status explicitly.

14 CHAPTER

Using I/O

The earlier chapters of this book have used parts of the C# I/O system, such as **Console.WriteLine( )**, but have done so without much formal explanation. Because the I/O system is built upon a hierarchy of classes, it was not possible to present its theory and details without first discussing classes, inheritance, and exceptions. Now it is time to examine I/O in detail. Because C# uses the I/O system and classes defined by the .NET Framework, a discussion of I/O under C# is also a discussion of the .NET I/O system, in general.

This chapter examines both console I/O and file I/O. Be forewarned that the I/O system is quite large. This chapter describes the most important and commonly used features.

C#'s I/O Is Built Upon Streams

C# programs perform I/O through streams. A *stream* is an abstraction that either produces or consumes information. A stream is linked to a physical device by the I/O system. All streams behave in the same manner, even if the actual physical devices they are linked to differ. Thus, the I/O classes and methods can be applied to many types of devices. For example, the same methods that you use to write to the console can also be used to write to a disk file.

Byte Streams and Character Streams

At the lowest level, all C# I/O operates on bytes. This makes sense because many devices are byte oriented when it comes to I/O operations. Frequently, though, we humans prefer to communicate using characters. Recall that in C#, **char** is a 16-bit type, and **byte** is an 8-bit type. If you are using the ASCII character set, then it is easy to convert between **char** and **byte**; just ignore the high-order byte of the **char** value. But this won't work for the rest of the Unicode characters, which need both bytes (and possibly more). Thus, byte streams are not perfectly suited to handling character-based I/O. To solve this problem, the .NET Framework defines several classes that convert a byte stream into a character stream, handling the translation of **byte**-to-**char** and **char**-to-**byte** automatically.

The Predefined Streams

Three predefined streams, which are exposed by the properties called **Console.In**, **Console.Out**, and **Console.Error**, are available to all programs that use the **System** namespace. **Console.Out** refers to the standard output stream. By default, this is the

console. When you call **Console.WriteLine( )**, for example, it automatically sends information to **Console.Out**. **Console.In** refers to standard input, which is, by default, the keyboard. **Console.Error** refers to the standard error stream, which is also the console by default. However, these streams can be redirected to any compatible I/O device. The standard streams are character streams. Thus, these streams read and write characters.

The Stream Classes

The .NET Framework defines both byte and character stream classes. However, the character stream classes are really just wrappers that convert an underlying byte stream to a character stream, handling any conversion automatically. Thus, the character streams, while logically separate, are built upon byte streams.

The core stream classes are defined within the **System.IO** namespace. To use these classes, you will usually include the following statement near the top of your program:

```
using System.IO;
```

The reason that you don't have to specify **System.IO** for console input and output is that the **Console** class is defined in the **System** namespace.

The Stream Class

The core stream class is **System.IO.Stream**. **Stream** represents a byte stream and is a base class for all other stream classes. It is also abstract, which means that you cannot instantiate a **Stream** object. **Stream** defines a set of standard stream operations. Table 14-1 shows several commonly used methods defined by **Stream**.

Several of the methods shown in Table 14-1 will throw an **IOException** if an I/O error occurs. If an invalid operation is attempted, such as attempting to write to a stream that is read-only, a **NotSupportedException** is thrown. Other exceptions are possible, depending on the specific method.

Method	Description
void Close( )	Closes the stream.
void Flush( )	Writes the contents of the stream to the physical device.
int ReadByte( )	Returns an integer representation of the next available byte of input. Returns –1 when the end of the file is encountered.
int Read(byte[ ] *buf*, int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buf* starting at *buf*[*offset*], returning the number of bytes successfully read.
long Seek(long *offset*, SeekOrigin *origin*)	Sets the current position in the stream to the specified *offset* from the specified *origin*. It returns the new position.
void WriteByte(byte *b*)	Writes a single byte to an output stream.
int Write(byte[ ] *buf*, int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buf*, beginning at *buf*[*offset*]. The number of bytes written is returned.

TABLE 14-1 Some of the Methods Defined by **Stream**

Notice that **Stream** defines methods that read and write data. However, not all streams will support both of these operations because it is possible to open read-only or write-only streams. Also, not all streams will support position requests via **Seek( )**. To determine the capabilities of a stream, you will use one or more of **Stream**'s properties. They are shown in Table 14-2. Also shown are the **Length** and **Position** properties, which contain the length of the stream and its current position.

The Byte Stream Classes

Several concrete byte streams are derived from **Stream**. Those defined in the **System.IO** namespace are shown here:

Stream Class	Description
BufferedStream	Wraps a byte stream and adds buffering. Buffering provides a performance enhancement in many cases.
FileStream	A byte stream designed for file I/O.
MemoryStream	A byte stream that uses memory for storage.
UnmanagedMemoryStream	A byte stream that uses unmanaged memory for storage.

Several other concrete stream classes are also supported by the .NET framework, which provide support for compressed files, sockets, and pipes, among others. It is also possible to derive your own stream classes. However, for the vast majority of applications, the built-in streams will be sufficient.

The Character Stream Wrapper Classes

To create a character stream, wrap a byte stream inside one of the character stream wrappers. At the top of the character stream hierarchy are the abstract classes **TextReader** and **TextWriter**. **TextReader** handles input and **TextWriter** handles output. The methods defined by these two abstract classes are available to all of their subclasses. Thus, they form a minimal set of I/O functions that all character streams will have.

Method	Description
bool CanRead	This property is true if the stream can be read. This property is read-only.
bool CanSeek	This property is true if the stream supports position requests. This property is read-only.
bool CanTimeout	This property is true if the stream can time out. This property is read-only.
bool CanWrite	This property is true if the stream can be written. This property is read-only.
long Length	This property contains the length of the stream. This property is read-only.
long Position	This property represents the current position of the stream. This property is read/write.
int ReadTimeout	This property represents the length of time before a time-out will occur for read operations. This property is read/write.
int WriteTimeout	This property represents the length of time before a time-out will occur for write operations. This property is read/write.

TABLE 14-2 The Properties Defined by **Stream**

Table 14-3 shows the input methods in **TextReader**. In general, these methods can throw an **IOException** on error. (Some can throw other types of exceptions, too.) Of particular interest is the **ReadLine( )** method, which reads an entire line of text, returning it as a **string**. This method is useful when reading input that contains embedded spaces.

TextWriter defines versions of **Write( )** and **WriteLine( )** that output all of the built-in types. For example, here are just a few of their overloaded versions:

Method	Description
void Write(int *val*)	Writes an **int**.
void Write(double *val*)	Writes a **double**.
void Write(bool *val*)	Writes a **bool**.
void WriteLine(string *val*)	Writes a **string** followed by a newline.
void WriteLine(uint *val)*	Writes a **uint** followed by a newline.
void WriteLine(char *val*)	Writes a character followed by a newline.

All throw an **IOException** if an error occurs while writing.

TextWriter also defines the **Close( )** and **Flush( )** methods shown here:

```
virtual void Close( )
virtual void Flush( )
```

Flush( ) causes any data remaining in the output buffer to be written to the physical medium. **Close( )** closes the stream.

Method	Description
int Peek( )	Obtains the next character from the input stream, but does not remove that character. Returns –1 if no character is available.
int Read( )	Returns an integer representation of the next available character from the invoking input stream. Returns –1 when the end of the stream is encountered.
int Read(char[ ] *buf*, int *offset*, int *numChars*)	Attempts to read up to *numChars* characters into *buf* starting at *buf*[*offset*], returning the number of characters successfully read.
int ReadBlock(char[ ] *buf*, int *offset*, int *numChars*)	Attempts to read up to *numChars* characters into *buf* starting at *buf*[*offset*], returning the number of characters successfully read.
string ReadLine( )	Reads the next line of text and returns it as a string. Null is returned if an attempt is made to read at end-of-file.
string ReadToEnd( )	Reads all of the remaining characters in a stream and returns them as a string.

Table 14-3 The Input Methods Defined by **TextReader**

The **TextReader** and **TextWriter** classes are implemented by several character-based stream classes, including those shown here. Thus, these streams provide the methods and properties specified by **TextReader** and **TextWriter**.

Stream Class	Description
StreamReader	Read characters from a byte stream. This class wraps a byte input stream.
StreamWriter	Write characters to a byte stream. This class wraps a byte output stream.
StringReader	Read characters from a string.
StringWriter	Write characters to a string.

Binary Streams

In addition to the byte and character streams, there are two binary stream classes that can be used to read and write binary data directly. These streams are called **BinaryReader** and **BinaryWriter**. We will look closely at these later in this chapter when binary file I/O is discussed.

Now that you understand the general layout of the I/O system, the rest of this chapter will examine its various pieces in detail, beginning with console I/O.

Console I/O

Console I/O is accomplished through the standard streams **Console.In**, **Console.Out**, and **Console.Error**. Console I/O has been used since Chapter 2, so you are already familiar with it. As you will see, it has some additional capabilities.

Before we begin, however, it is important to emphasize a point made earlier in this book: Most real applications of C# will not be text-based, console programs. Rather, they will be graphically oriented programs or components that rely upon a windowed interface for interaction with the user, or will be server-side code. Thus, the portion of the I/O system that relates to console input and output is not widely used. Although text-based programs are excellent as teaching examples, for short utility programs, and for some types of components, they are not suitable for most real-world applications.

Reading Console Input

Console.In is an instance of **TextReader**, and you can use the methods and properties defined by **TextReader** to access it. However, you will generally use the methods provided by **Console**, which automatically read from **Console.In**. **Console** defines three input methods. The first two, **Read( )** and **ReadLine( )**, have been available since .NET Framework 1.0. The third, **ReadKey( )**, was added by .NET Framework 2.0.

To read a single character, use the **Read( )** method:

```
static int Read( )
```

Read( ) returns the next character read from the console. It waits until the user presses a key and then returns the result. The character is returned as an **int**, which must be cast to **char**. **Read( )** returns –1 on error. This method will throw an **IOException** on failure. When using **Read( )**, console input is line-buffered, so you must press ENTER before any character that you type will be sent to your program.

Here is a program that reads a character from the keyboard using **Read( )**:

```
// Read a character from the keyboard.

using System;

class KbIn {
  static void Main() {
    char ch;

    Console.Write("Press a key followed by ENTER: ");

    ch = (char) Console.Read(); // get a char

    Console.WriteLine("Your key is: " + ch);
  }
}
```

Here is a sample run:

```
Press a key followed by ENTER: t
Your key is: t
```

The fact that **Read( )** is line-buffered is a source of annoyance at times. When you press ENTER, a carriage-return, line-feed sequence is entered into the input stream. Furthermore, these characters are left pending in the input buffer until you read them. Thus, for some applications, you may need to remove them (by reading them) before the next input operation. (To read keystrokes from the console in a non-line-buffered manner, you can use **ReadKey( )**, described later in this section.)

To read a string of characters, use the **ReadLine( )** method. It is shown here:

static string ReadLine( )

ReadLine( ) reads characters until you press ENTER and returns them in a **string** object. This method will also throw an **IOException** on failure.

Here is a program that demonstrates reading a string from **Console.In** by using **ReadLine( )**:

```
// Input from the console using ReadLine().

using System;

class ReadString {
  static void Main() {
    string str;

    Console.WriteLine("Enter some characters.");
    str = Console.ReadLine();
    Console.WriteLine("You entered: " + str);
  }
}
```

Here is a sample run:

```
Enter some characters.
This is a test.
You entered: This is a test.
```

Although the **Console** methods are the easiest way to read from **Console.In**, you can call methods on the underlying **TextReader**. For example, here is the preceding program rewritten to use the **ReadLine( )** method defined by **TextReader**:

```
// Read a string from the keyboard, using Console.In directly.

using System;

class ReadChars2 {
  static void Main() {
    string str;

    Console.WriteLine("Enter some characters.");

    str = Console.In.ReadLine(); // call TextReader's ReadLine() method

    Console.WriteLine("You entered: " + str);
  }
}
```

Notice how **ReadLine( )** is now invoked directly on **Console.In**. The key point here is that if you need access to the methods defined by the **TextReader** that underlies **Console.In**, you will invoke those methods as shown in this example.

Using ReadKey( )

Beginning with version 2.0, the .NET Framework has included a method in **Console** that enables you to read individual keystrokes directly from the keyboard in a non-line-buffered manner. This method is called **ReadKey( )**. When it is called, it waits until a key is pressed. When a key is pressed, **ReadKey( )** returns the keystroke immediately. The user does not need to press ENTER. Thus, **ReadKey( )** allows keystrokes to be read and processed in real time.

ReadKey( ) has these two forms.

static ConsoleKeyInfo ReadKey( )

static ConsoleKeyInfo ReadKey(bool *noDisplay*)

The first form waits for a key to be pressed. When that occurs, it returns the key and also displays the key on the screen. The second form also waits for and returns a keypress. However, if *noDisplay* is true, then the key is not displayed. If *noDisplay* is false, the key is displayed.

ReadKey( ) returns information about the keypress in an object of type **ConsoleKeyInfo**, which is a structure. It contains the following read-only properties.

char KeyChar

ConsoleKey Key

ConsoleModifiers Modifiers

KeyChar contains the **char** equivalent of the character that was pressed. **Key** contains a value from the **ConsoleKey** enumeration, which is an enumeration of all the keys on the keyboard. **Modifiers** describes which, if any, of the keyboard modifiers ATL, CTRL, or SHIFT were pressed when the keystroke was generated. These modifiers are represented by the **ConsoleModifiers** enumeration, which has these values: **Control**, **Shift**, and **Alt**. More than one modifier value might be present in **Modifiers**.

The major advantage to **ReadKey( )** is that it provides a means of achieving interactive keyboard input because it is not line buffered. To see the effect of this, try the following program:

```
// Read keystrokes from the console by using ReadKey().

using System;

class ReadKeys {
  static void Main() {
    ConsoleKeyInfo keypress;

    Console.WriteLine("Enter keystrokes. Enter Q to stop.");

    do {
      keypress = Console.ReadKey(); // read keystrokes

      Console.WriteLine(" Your key is: " + keypress.KeyChar);

      // Check for modifier keys.
      if((ConsoleModifiers.Alt & keypress.Modifiers) != 0)
        Console.WriteLine("Alt key pressed.");
      if((ConsoleModifiers.Control & keypress.Modifiers) != 0)
        Console.WriteLine("Control key pressed.");
      if((ConsoleModifiers.Shift & keypress.Modifiers) != 0)
        Console.WriteLine("Shift key pressed.");

    } while(keypress.KeyChar != 'Q');
  }
}
```

A sample run is shown here:

```
Enter keystrokes. Enter Q to stop.
a Your key is: a
b Your key is: b
d Your key is: d
A Your key is: A
Shift key pressed.
B Your key is: B
Shift key pressed.
C Your key is: C
Shift key pressed.
• Your key is: •
Control key pressed.
Q Your key is: Q
Shift key pressed.
```

As the output confirms, each time a key is pressed, **ReadKey( )** immediately returns the keypress. As explained, this differs from **Read( )** and **ReadLine( )**, which use line-buffered input. Therefore, if you want to achieve interactive responses from the keyboard, use **ReadKey( )**.

Writing Console Output

Console.Out and **Console.Error** are objects of type **TextWriter**. Console output is most easily accomplished with **Write( )** and **WriteLine( )**, with which you are already familiar. Versions of these methods exist that output for each of the built-in types. **Console** defines its own versions of **Write( )** and **WriteLine( )** so they can be called directly on **Console**, as you have been doing throughout this book. However, you can invoke these (and other) methods on the **TextWriter** that underlies **Console.Out** and **Console.Error** if you choose.

Here is a program that demonstrates writing to **Console.Out** and **Console.Error**. By default, both write to the console.

```
// Write to Console.Out and Console.Error.

using System;

class ErrOut {
  static void Main() {
    int a=10, b=0;
    int result;

    Console.Out.WriteLine("This will generate an exception.");
    try {
      result = a / b; // generate an exception
    } catch(DivideByZeroException exc) {
      Console.Error.WriteLine(exc.Message);
    }
  }
}
```

The output from the program is shown here:

```
This will generate an exception.
Attempted to divide by zero.
```

Sometimes newcomers to programming are confused about when to use **Console.Error**. Since both **Console.Out** and **Console.Error** default to writing their output to the console, why are there two different streams? The answer lies in the fact that the standard streams can be redirected to other devices. For example, **Console.Error** can be redirected to write to a disk file, rather than the screen. Thus, it is possible to direct error output to a log file, for example, without affecting console output. Conversely, if console output is redirected and error output is not, then error messages will appear on the console, where they can be seen. We will examine redirection later, after file I/O has been described.

FileStream and Byte-Oriented File I/O

The .NET Framework provides classes that allow you to read and write files. Of course, the most common type of files are disk files. At the operating system level, all files are byte

oriented. As you would expect, there are methods to read and write bytes from and to a file. Thus, reading and writing files using byte streams is very common. You can also wrap a byte-oriented file stream within a character-based object. Character-based file operations are useful when text is being stored. Character streams are discussed later in this chapter. Byte-oriented I/O is described here.

To create a byte-oriented stream attached to a file, you will use the **FileStream** class. **FileStream** is derived from **Stream** and contains all of **Stream**'s functionality.

Remember, the stream classes, including **FileStream**, are defined in **System.IO**. Thus, you will usually include

```
using System.IO;
```

near the top of any program that uses them.

Opening and Closing a File

To create a byte stream linked to a file, create a **FileStream** object. **FileStream** defines several constructors. Perhaps its most commonly used is the one shown here:

FileStream(string *filename*, FileMode *mode*)

Here, *filename* specifies the name of the file to open, which can include a full path specification. The *mode* parameter specifies how the file will be opened. It must be one of the values defined by the **FileMode** enumeration. These values are shown in Table 14-4. In general, this constructor opens a file for read/write access. The exception is when the file is opened using **FileMode.Append**. In this case, the file is write-only.

If a failure occurs when attempting to open the file, an exception will be thrown. If the file cannot be opened because it does not exist, **FileNotFoundException** will be thrown. If the file cannot be opened because of some type of I/O error, **IOException** will be thrown. Other possible exceptions are **ArgumentNullException** (the filename is null), **ArgumentException** (the filename is invalid), **ArgumentOutOfRangeException** (the mode is invalid), **SecurityException** (user does not have access rights), **PathTooLongException** (the filename/path is too long), **NotSupportedException** (the filename specifies an unsupported device), and **DirectoryNotFoundException** (specified directory is invalid).

Value	Description
FileMode.Append	Output is appended to the end of file.
FileMode.Create	Creates a new output file. Any preexisting file by the same name will be destroyed.
FileMode.CreateNew	Creates a new output file. The file must not already exist.
FileMode.Open	Opens a preexisting file.
FileMode.OpenOrCreate	Opens a file if it exists, or creates the file if it does not already exist.
FileMode.Truncate	Opens a preexisting file, but reduces its length to zero.

TABLE 14-4 The **FileMode** Values

The exceptions **PathTooLongException**, **DirectoryNotFoundException**, and **FileNotFoundException** are subclasses of **IOException**. Thus, it is possible to catch all three by catching **IOException**.

The following shows one way to open the file **test.dat** for input:

```
FileStream fin;

try {
  fin = new FileStream("test", FileMode.Open);
}
catch(IOException exc) { // catch all I/O exceptions
  Console.WriteLine(exc.Message);
  // Handle the error.
}
catch(Exception exc { // catch any other exception.
  Console.WriteLine(exc.Message);
  // Handle the error.
}
```

Here, the first **catch** clause handles situations in which the file is not found, the path is too long, the directory does not exist, or other I/O errors occur. The second **catch**, which is a "catch all" clause for all other types of exceptions, handles the other possible errors (possibly by rethrowing the exception). You could also check for each error individually, reporting more specifically the problem that occurred and taking remedial action specific to that error.

For the sake of simplicity, the examples in this book will catch only **IOException**, but your real-world code may (probably will) need to handle the other possible exceptions, depending upon the circumstances. Also, the exception handlers in this chapter simply report the error, but in many cases, your code should take steps to correct the problem when possible. For example, you might reprompt the user for a filename if the one previously entered is not found.

Remember *To keep the code simple, the examples in this chapter catch only* **IOException**, *but your own code may need to handle other possible exceptions or handle each type of I/O exception individually.*

As mentioned, the **FileStream** constructor just described opens a file that has read/write access. If you want to restrict access to just reading or just writing, use this constructor instead:

FileStream(string *filename*, FileMode *mode*, FileAccess *how*)

As before, *filename* specifies the name of the file to open, and *mode* specifies how the file will be opened. The value passed in *how* determines how the file can be accessed. It must be one of the values defined by the **FileAccess** enumeration, which are shown here:

FileAccess.Read	FileAccess.Write	FileAccess.ReadWrite

For example, this opens a read-only file:

```
FileStream fin = new FileStream("test.dat", FileMode.Open, FileAccess.Read);
```

When you are done with a file, you should close it by calling **Close( )**. Its general form is shown here:

void Close( )

Closing a file releases the system resources allocated to the file, allowing them to be used by another file. As a point of interest, **Close( )** works by calling **Dispose( )**, which actually frees the resources.

NOTE *The* **using** *statement, described in Chapter 20, offers a way to automatically close a file when it is no longer needed. This approach is beneficial in many file-handling situations because it provides a simple means to ensure that a file is closed when it is no longer needed. However, to clearly illustrate the fundamentals of file handling, including the point at which a file can be closed, this chapter explicitly calls* **Close( )** *in all cases.*

Reading Bytes from a FileStream

FileStream defines two methods that read bytes from a file: **ReadByte( )** and **Read( )**. To read a single byte from a file, use **ReadByte( )**, whose general form is shown here:

int ReadByte( )

Each time it is called, it reads a single byte from the file and returns it as an integer value. It returns –1 when the end of the file is encountered. Possible exceptions include **NotSupportedException** (the stream is not opened for input) and **ObjectDisposedException** (the stream is closed).

To read a block of bytes, use **Read( )**, which has this general form:

int Read(byte[ ] *buf*, int *offset*, int *numBytes*)

Read( ) attempts to read up to *numBytes* bytes into *buf* starting at *buf*[*offset*]. It returns the number of bytes successfully read. An **IOException** is thrown if an I/O error occurs. Several other types of exceptions are possible, including **NotSupportedException**, which is thrown if reading is not supported by the stream.

The following program uses **ReadByte( )** to input and display the contents of a text file, the name of which is specified as a command-line argument. Note the program handles two errors that might occur when this program is first executed: the specified file not being found or the user forgetting to include the name of the file.

```
/* Display a text file.

   To use this program, specify the name of the file that you
   want to see. For example, to see a file called TEST.CS,
   use the following command line.

   ShowFile TEST.CS
*/

using System;
using System.IO;

class ShowFile {
  static void Main(string[] args) {
```

```
    int i;
    FileStream fin;

    if(args.Length != 1) {
      Console.WriteLine("Usage: ShowFile File");
      return;
    }

    try {
      fin = new FileStream(args[0], FileMode.Open);
    } catch(IOException exc) {
      Console.WriteLine("Cannot Open File");
      Console.WriteLine(exc.Message);
      return;
    }

    // Read bytes until EOF is encountered.
    do {
      try {
        i = fin.ReadByte();
      } catch(IOException exc) {
        Console.WriteLine("Error Reading File");
        Console.WriteLine(exc.Message);
        break;
      }
      if(i != -1) Console.Write((char) i);
    } while(i != -1);

    fin.Close();
  }
}
```

Writing to a File

To write a byte to a file, use the **WriteByte( )** method. Its simplest form is shown here:

void WriteByte(byte *val*)

This method writes the byte specified by *val* to the file. If the underlying stream is not opened for output, a **NotSupportedException** is thrown. If the stream is closed, **ObjectDisposedException** is thrown.

You can write an array of bytes to a file by calling **Write( )**. It is shown here:

void Write(byte[ ] *buf*, int *offset*, int *numBytes*)

Write( ) writes *numBytes* bytes from the array *buf*, beginning at *buf*[*offset*], to the file. The number of bytes written is returned. If an error occurs during writing, an **IOException** is thrown. If the underlying stream is not opened for output, a **NotSupportedException** is thrown. Several other exceptions are also possible.

As you may know, when file output is performed, often that output is not immediately written to the actual physical device. Instead, output is buffered by the operating system until a sizable chunk of data can be written all at once. This improves the efficiency of the system. For example, disk files are organized by sectors, which might be anywhere from 128 bytes long, on up. Output is usually buffered until an entire sector can be written all at once.

However, if you want to cause data to be written to the physical device whether the buffer is full or not, you can call **Flush( )**, shown here:

```
void Flush( )
```

An **IOException** is thrown on failure. If the stream is closed, **ObjectDisposedException** is thrown.

Once you are done with an output file, you must remember to close it using **Close( )**. Doing so ensures that any output remaining in a disk buffer is actually written to the disk. Thus, there is no reason to call **Flush( )** before closing a file.

Here is a simple example that writes to a file:

```
// Write to a file.

using System;
using System.IO;

class WriteToFile {
  static void Main(string[] args) {
    FileStream fout;

    // Open output file.
    try {
      fout = new FileStream("test.txt", FileMode.Create);
    } catch(IOException exc) {
      Console.WriteLine("Cannot Open File");
      Console.WriteLine(exc.Message);
      return;
    }

    // Write the alphabet to the file.
    try {
      for(char c = 'A'; c <= 'Z'; c++)
        fout.WriteByte((byte) c);
    } catch(IOException exc) {
      Console.WriteLine("Error Writing File");
      Console.WriteLine(exc.Message);
    }

    fout.Close();
  }
}
```

The program first opens a file called **test.txt** for output. It then writes the uppercase alphabet to the file. Finally, it closes the file. Notice how possible I/O errors are handled by the **try**/**catch** blocks. After this program executes, **test.txt** will contain the following output:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Using FileStream to Copy a File

One advantage to the byte-oriented I/O used by **FileStream** is that you can use it on any type of file—not just those that contain text. For example, the following program copies any type of file, including executable files. The names of the source and destination files are specified on the command line.

```
/* Copy a file.

   To use this program, specify the name of the source
   file and the destination file. For example, to copy a
   file called FIRST.DAT to a file called SECOND.DAT, use
   the following command line:

   CopyFile FIRST.DAT SECOND.DAT
*/

using System;
using System.IO;

class CopyFile {
  static void Main(string[] args) {
    int i;
    FileStream fin;
    FileStream fout;

    if(args.Length != 2) {
      Console.WriteLine("Usage: CopyFile From To");
      return;
    }

    // Open input file.
    try {
      fin = new FileStream(args[0], FileMode.Open);
    } catch(IOException exc) {
      Console.WriteLine("Cannot Open Input File");
      Console.WriteLine(exc.Message);
      return;
    }

    // Open output file.
    try {
      fout = new FileStream(args[1], FileMode.Create);
    } catch(IOException exc) {
      Console.WriteLine("Cannot Open Output File");
      Console.WriteLine(exc.Message);
      fin.Close();
      return;
    }

    // Copy File
    try {
      do {
        i = fin.ReadByte();
        if(i != -1) fout.WriteByte((byte)i);
      } while(i != -1);
    } catch(IOException exc) {
      Console.WriteLine("Error Copying File");
      Console.WriteLine(exc.Message);
    }

    fin.Close();
```

```
    fout.Close();
  }
}
```

Character-Based File I/O

Although byte-oriented file handling is quite common, it is possible to use character-based streams for this purpose. The advantage to the character streams is that they operate directly on Unicode characters. Thus, if you want to store Unicode text, the character streams are certainly your best option. In general, to perform character-based file operations, you will wrap a **FileStream** inside either a **StreamReader** or a **StreamWriter**. These classes automatically convert a byte stream into a character stream, and vice versa.

Remember, at the operating system level, a file consists of a set of bytes. Using a **StreamReader** or **StreamWriter** does not alter this fact.

StreamWriter is derived from **TextWriter**. **StreamReader** is derived from **TextReader**. Thus, **StreamWriter** and **StreamReader** have access to the methods and properties defined by their base classes.

Using StreamWriter

To create a character-based output stream, wrap a **Stream** object (such as a **FileStream**) inside a **StreamWriter**. **StreamWriter** defines several constructors. One of its most popular is shown here:

StreamWriter(Stream *stream*)

Here, *stream* is the name of an open stream. This constructor throws an **ArgumentException** if *stream* is not opened for output and an **ArgumentNullException** if *stream* is null. Once created, a **StreamWriter** automatically handles the conversion of characters to bytes.

Here is a simple key-to-disk utility that reads lines of text entered at the keyboard and writes them to a file called "test.txt." Text is read until the user enters the word "stop". It uses a **FileStream** wrapped in a **StreamWriter** to output to the file.

```
// A simple key-to-disk utility that demonstrates a StreamWriter.

using System;
using System.IO;

class KtoD {
  static void Main() {
    string str;
    FileStream fout;

    try {
      fout = new FileStream("test.txt", FileMode.Create);
    }
    catch(IOException exc) {
      Console.WriteLine("Cannot Open File");
      Console.WriteLine(exc.Message);
      return ;
    }
    StreamWriter fstr_out = new StreamWriter(fout);
```

```
    Console.WriteLine("Enter text ('stop' to quit).");
    do {
      Console.Write(": ");
      str = Console.ReadLine();

      if(str != "stop") {
        str = str + "\r\n"; // add newline
        try {
          fstr_out.Write(str);
          } catch(IOException exc) {
             Console.WriteLine("Error Writing File");
             Console.WriteLine(exc.Message);
             break;
          }
      }
    } while(str != "stop");

    fstr_out.Close();
  }
}
```

In some cases, it might be more convenient to open a file directly using **StreamWriter**. To do so, use one of these constructors:

StreamWriter(string *filename*)
StreamWriter(string *filename*, bool *appendFlag*)

Here, *filename* specifies the name of the file to open, which can include a full path specifier. In the second form, if *appendFlag* is true, then output is appended to the end of an existing file. Otherwise, output overwrites the specified file. In both cases, if the file does not exist, it is created. Also, both throw an **IOException** if an I/O error occurs. Other exceptions are also possible.

Here is the key-to-disk program rewritten so that it uses **StreamWriter** to open the output file:

```
// Open a file using StreamWriter.

using System;
using System.IO;

class KtoD {
  static void Main() {
    string str;
    StreamWriter fstr_out;

    try {
      fstr_out = new StreamWriter("test.txt");
    }
    catch(IOException exc) {
      Console.WriteLine("Cannot Open File");
      Console.WriteLine(exc.Message);
      return ;
    }

    Console.WriteLine("Enter text ('stop' to quit).");
```

```
    do {
      Console.Write(": ");
      str = Console.ReadLine();

      if(str != "stop") {
        str = str + "\r\n"; // add newline
        try {
          fstr_out.Write(str);
        } catch(IOException exc) {
          Console.WriteLine("Error Writing File");
          Console.WriteLine(exc.Message);
          break;
        }
      }
    } while(str != "stop");

    fstr_out.Close();
  }
}
```

Using a StreamReader

To create a character-based input stream, wrap a byte stream inside a **StreamReader**. **StreamReader** defines several constructors. A frequently used one is shown here:

StreamReader(Stream *stream*)

Here, *stream* is the name of an open stream. This constructor throws an **ArgumentNullException** if *stream* is null. It throws **ArgumentException** if *stream* is not opened for input. Once created, a **StreamReader** will automatically handle the conversion of bytes to characters.

The following program creates a simple disk-to-screen utility that reads a text file called "test.txt" and displays its contents on the screen. Thus, it is the complement of the key-to-disk utility shown in the previous section:

```
// A simple disk-to-screen utility that demonstrates a StreamReader.

using System;
using System.IO;

class DtoS {
  static void Main() {
    FileStream fin;
    string s;

    try {
      fin = new FileStream("test.txt", FileMode.Open);
    }
    catch(IOException exc) {
      Console.WriteLine("Error Opening File");
      Console.WriteLine(exc.Message);
      return ;
    }

    StreamReader fstr_in = new StreamReader(fin);
```

```
    try {
      while((s = fstr_in.ReadLine()) != null) {
        Console.WriteLine(s);
      }
    } catch(IOException exc) {
      Console.WriteLine("Error Reading File");
      Console.WriteLine(exc.Message);
    }

    fstr_in.Close();
  }
}
```

In the program, notice how the end of the file is determined. When the reference returned by **ReadLine( )** is null, the end of the file has been reached. Although this approach works, **StreamReader** provides an alternative means of detecting the end of the stream: the **EndOfStream** property. This read-only property is true when the end of the stream has been reached and false otherwise. Therefore, you can use **EndOfStream** to watch for the end of a file. For example, here is another way to write the **while** loop that reads the file:

```
while(!fstr_in.EndOfStream) {
  s = fstr_in.ReadLine();
  Console.WriteLine(s);
}
```

In this case, the use of **EndOfStream** makes the code a bit easier to understand but does not change the overall structure of the sequence. There are times, however, when the use of **EndOfStream** can simplify an otherwise tricky situation, adding clarity and improving structure.

As with **StreamWriter**, in some cases, you might find it easier to open a file directly using **StreamReader**. To do so, use this constructor:

StreamReader(string *filename*)

Here, *filename* specifies the name of the file to open, which can include a full path specifier. The file must exist. If it doesn't, a **FileNotFoundException** is thrown. If *filename* is null, then an **ArgumentNullException** is thrown. If *filename* is an empty string, **ArgumentException** is thrown. **IOException** and **DirectoryNotFoundException** are also possible.

Redirecting the Standard Streams

As mentioned earlier, the standard streams, such as **Console.In**, can be redirected. By far, the most common redirection is to a file. When a standard stream is redirected, input and/or output is automatically directed to the new stream, bypassing the default devices. By redirecting the standard streams, your program can read commands from a disk file, create log files, or even read input from a network connection.

Redirection of the standard streams can be accomplished in two ways. First, when you execute a program on the command line, you can use the **<** and **>** operators to redirect **Console.In** and/or **Console.Out**, respectively. For example, given this program:

```
using System;
```

```
class Test {
  static void Main() {
    Console.WriteLine("This is a test.");
  }
}
```

executing the program like this

```
Test > log
```

will cause the line "This is a test." to be written to a file called **log**. Input can be redirected in the same way. The thing to remember when input is redirected is that you must make sure that what you specify as an input source contains sufficient input to satisfy the demands of the program. If it doesn't, the program will hang.

The **<** and **>** command-line redirection operators are not part of C#, but are provided by the operating system. Thus, if your environment supports I/O redirection (as is the case with Windows), you can redirect standard input and standard output without making any changes to your program. However, there is a second way that you can redirect the standard streams that is under program control. To do so, you will use the **SetIn( )**, **SetOut( )**, and **SetError( )** methods, shown here, which are members of **Console**:

```
static void SetIn(TextReader input)
static void SetOut(TextWriter output)
static void SetError(TextWriter output)
```

Thus, to redirect input, call **SetIn( )**, specifying the desired stream. You can use any input stream as long as it is derived from **TextReader**. To redirect output, call **SetOut ( )**, specifying the desired output stream, which must be derived from **TextWriter**. For example, to redirect output to a file, specify a **FileStream** that is wrapped in a **StreamWriter**. The following program shows an example:

```
// Redirect Console.Out.

using System;
using System.IO;

class Redirect {
  static void Main() {
    StreamWriter log_out;

    try {
      log_out = new StreamWriter("logfile.txt");
    }
    catch(IOException exc) {
      Console.WriteLine("Error Opening Log File");
      Console.WriteLine(exc.Message);
      return ;
    }

    // Redirect standard out to logfile.txt.
    Console.SetOut(log_out);
```

```
    try {
      Console.WriteLine("This is the start of the log file.");

      for(int i=0; i<10; i++) Console.WriteLine(i);

      Console.WriteLine("This is the end of the log file.");
    } catch(IOException exc) {
        Console.WriteLine("Error Writing Log File");
        Console.WriteLine(exc.Message);
    }

    log_out.Close();
  }
}
```

When you run this program, you won't see any of the output on the screen, but the file **logfile.txt** will contain the following:

```
This is the start of the log file.
0
1
2
3
4
5
6
7
8
9
This is the end of the log file.
```

On your own, you might want to experiment with redirecting the other built-in streams.

Reading and Writing Binary Data

So far, we have just been reading and writing bytes or characters, but it is possible—indeed, common—to read and write other types of data. For example, you might want to create a file that contains the **int**s, **double**s, or **short**s. To read and write binary values of the C# built-in types, you will use **BinaryReader** and **BinaryWriter**. When using these streams, it is important to understand that this data is read and written using its internal, binary format, not its human-readable text form.

BinaryWriter

A **BinaryWriter** is a wrapper around a byte stream that manages the writing of binary data. Its most commonly used constructor is shown here:

BinaryWriter(Stream *outputStream*)

Here, *outputStream* is the stream to which data is written. To write output to a file, you can use the object created by **FileStream** for this parameter. If *outputStream* is null, then an **ArgumentNullException** is thrown. If *outputStream* has not been opened for writing, **ArgumentException** is thrown.

BinaryWriter defines methods that can write all of C#'s built-in types. Several are shown in Table 14-5. Notice that a **string** is written using its internal format, which includes a length specifier. **BinaryWriter** also defines the standard **Close( )** and **Flush( )** methods, which work as described earlier.

BinaryReader

A **BinaryReader** is a wrapper around a byte stream that handles the reading of binary data. Its most commonly used constructor is shown here:

BinaryReader(Stream *inputStream*)

Here, *inputStream* is the stream from which data is read. To read from a file, you can use the object created by **FileStream** for this parameter. If *inputStream* has not been opened for reading or is otherwise invalid, **ArgumentException** is thrown.

BinaryReader provides methods for reading all of C#'s simple types. Several commonly used methods are shown in Table 14-6. Notice that **ReadString( )** reads a **string** that is stored using its internal format, which includes a length specifier. These methods throw an **IOException** if an error occurs. (Other exceptions are also possible.)

Method	Description
void Write(sbyte *val*)	Writes a signed byte.
void Write(byte *val*)	Writes an unsigned byte.
void Write(byte[ ] *buf*)	Writes an array of bytes.
void Write(short *val*)	Writes a short integer.
void Write(ushort *val*)	Writes an unsigned short integer.
void Write(int *val*)	Writes an integer.
void Write(uint *val*)	Writes an unsigned integer.
void Write(long *val*)	Writes a long integer.
void Write(ulong *val*)	Writes an unsigned long integer.
void Write(float *val*)	Writes a **float**.
void Write(double *val*)	Writes a **double**.
void Write(char *val*)	Writes a character.
void Write(char[ ] *buf*)	Writes an array of characters.
void Write(string *val*)	Writes a **string** using its internal representation, which includes a length specifier.

Table 14-5 Commonly Used Output Methods Defined by **BinaryWriter**

BinaryReader also defines three versions of **Read( )**, which are shown here:

Method	Description
int Read( )	Returns an integer representation of the next available character from the invoking input stream. Returns –1 when attempting to read at the end of the file.
int Read(byte[ ] *buf*, int *offset*, int *num*)	Attempts to read up to *num* bytes into *buf* starting at *buf*[*offset*], returning the number of bytes successfully read.
int Read(char[ ] *buf*, int *offset*, int *num*)	Attempts to read up to *num* characters into *buf* starting at *buf*[*offset*], returning the number of characters successfully read.

These methods will throw an **IOException** on failure. Other exceptions are possible. Also defined is the standard **Close( )** method.

Method	Description
bool ReadBoolean( )	Reads a **bool**.
byte ReadByte( )	Reads a **byte**.
sbyte ReadSByte( )	Reads an **sbyte**.
byte[ ] ReadBytes(int *num*)	Reads *num* bytes and returns them as an array.
char ReadChar( )	Reads a **char**.
char[ ] ReadChars(int *num*)	Reads *num* characters and returns them as an array.
double ReadDouble( )	Reads a **double**.
float ReadSingle( )	Reads a **float**.
short ReadInt16( )	Reads a **short**.
int ReadInt32( )	Reads an **int**.
long ReadInt64( )	Reads a **long**.
ushort ReadUInt16( )	Reads a **ushort**.
uint ReadUInt32( )	Reads a **uint**.
ulong ReadUInt64( )	Reads a **ulong**.
string ReadString( )	Reads a **string** that is represented in its internal, binary format, which includes a length specifier. This method should only be used to read a string that has been written using a **BinaryWriter**.

Table 14-6 Commonly Used Input Methods Defined by **BinaryReader**

Demonstrating Binary I/O

Here is a program that demonstrates **BinaryReader** and **BinaryWriter**. It writes and then reads back various types of data to and from a file.

```
// Write and then read back binary data.

using System;
using System.IO;

class RWData {
  static void Main() {
    BinaryWriter dataOut;
    BinaryReader dataIn;

    int i = 10;
    double d = 1023.56;
    bool b = true;
    string str = "This is a test";
    try {
      dataOut = new
        BinaryWriter(new FileStream("testdata", FileMode.Create));
    }
    catch(IOException exc) {
      Console.WriteLine("Cannot Open File For Output");
      Console.WriteLine(exc.Message);
      return;
    }

    // Write data to a file.
    try {
      Console.WriteLine("Writing " + i);
      dataOut.Write(i);

      Console.WriteLine("Writing " + d);
      dataOut.Write(d);

      Console.WriteLine("Writing " + b);
      dataOut.Write(b);

      Console.WriteLine("Writing " + 12.2 * 7.4);
      dataOut.Write(12.2 * 7.4);

      Console.WriteLine("Writing " + str);
      dataOut.Write(str);
    }
    catch(IOException exc) {
      Console.WriteLine("Error Writing File");
      Console.WriteLine(exc.Message);
    }

    dataOut.Close();

    Console.WriteLine();
```

```
    // Now, read the data.
    try {
      dataIn = new
          BinaryReader(new FileStream("testdata", FileMode.Open));
    }
    catch(IOException exc) {
      Console.WriteLine("Cannot Open File For Input");
      Console.WriteLine(exc.Message);
      return;
    }

    try {
      i = dataIn.ReadInt32();
      Console.WriteLine("Reading " + i);

      d = dataIn.ReadDouble();
      Console.WriteLine("Reading " + d);

      b = dataIn.ReadBoolean();
      Console.WriteLine("Reading " + b);

      d = dataIn.ReadDouble();
      Console.WriteLine("Reading " + d);

      str = dataIn.ReadString();
      Console.WriteLine("Reading " + str);
    }
    catch(IOException exc) {
      Console.WriteLine("Error Reading File");
      Console.WriteLine(exc.Message);
    }

    dataIn.Close();
  }
}
```

The output from the program is shown here:

```
Writing 10
Writing 1023.56
Writing True
Writing 90.28
Writing This is a test

Reading 10
Reading 1023.56
Reading True
Reading 90.28
Reading This is a test
```

If you examine the **testdata** file produced by this program, you will find that it contains binary data, not human-readable text.

Here is a more practical example that shows how powerful binary I/O is. The following program implements a very simple inventory program. For each item in the inventory, the

program stores the item's name, the number on hand, and its cost. Next, the program prompts the user for the name of an item. It then searches the database. If the item is found, the inventory information is displayed.

```
/* Use BinaryReader and BinaryWriter to implement
   a simple inventory program. */

using System;
using System.IO;

class Inventory {
  static void Main() {
    BinaryWriter dataOut;
    BinaryReader dataIn;

    string item; // name of item
    int onhand;  // number on hand
    double cost; // cost

    try {
      dataOut = new
        BinaryWriter(new FileStream("inventory.dat",
                                    FileMode.Create));
    }
    catch(IOException exc) {
      Console.WriteLine("Cannot Open Inventory File For Output");
      Console.WriteLine(exc.Message);
      return;
    }

    // Write some inventory data to the file.
    try {
      dataOut.Write("Hammers");
      dataOut.Write(10);
      dataOut.Write(3.95);

      dataOut.Write("Screwdrivers");
      dataOut.Write(18);
      dataOut.Write(1.50);

      dataOut.Write("Pliers");
      dataOut.Write(5);
      dataOut.Write(4.95);

      dataOut.Write("Saws");
      dataOut.Write(8);
      dataOut.Write(8.95);
    }
    catch(IOException exc) {
      Console.WriteLine("Error Writing Inventory File");
      Console.WriteLine(exc.Message);
    }

    dataOut.Close();
```

```
    Console.WriteLine();

    // Now, open inventory file for reading.
    try {
      dataIn = new
          BinaryReader(new FileStream("inventory.dat",
                       FileMode.Open));
    }
    catch(IOException exc) {
      Console.WriteLine("Cannot Open Inventory File For Input");
      Console.WriteLine(exc.Message);
      return;
    }

    // Lookup item entered by user.
    Console.Write("Enter item to lookup: ");
    string what = Console.ReadLine();
    Console.WriteLine();

    try {
      for(;;) {
        // Read an inventory entry.
        item = dataIn.ReadString();
        onhand = dataIn.ReadInt32();
        cost = dataIn.ReadDouble();

        // See if the item matches the one requested.
        // If so, display information.
        if(item.CompareTo(what) == 0) {
          Console.WriteLine(onhand + " " + item + " on hand. " +
                            "Cost: {0:C} each", cost);
          Console.WriteLine("Total value of {0}: {1:C}." ,
                            item, cost * onhand);
          break;
        }
      }
    }
    catch(EndOfStreamException) {
      Console.WriteLine("Item not found.");
    }
    catch(IOException exc) {
      Console.WriteLine("Error Reading Inventory File");
      Console.WriteLine(exc.Message);
    }

    dataIn.Close();
  }
}
```

Here is a sample run:

```
Enter item to look up: Screwdrivers

18 Screwdrivers on hand. Cost: $1.50 each
Total value of Screwdrivers: $27.00.
```

In the program, notice how inventory information is stored in its binary format. Thus, the number of items on hand and the cost is stored using their binary format rather than their human-readable text-based equivalents. This makes it is possible to perform computations on the numeric data without having to convert it from its human-readable form.

There is one other point of interest in the inventory program. Notice how the end of the file is detected. Since the binary input methods throw an **EndOfStreamException** when the end of the stream is reached, the program simply reads the file until either it finds the desired item or this exception is generated. Thus, no special mechanism is needed to detect the end of the file.

Random Access Files

Up to this point, we have been using *sequential files,* which are files that are accessed in a strictly linear fashion, one byte after another. However, you can also access the contents of a file in random order. One way to do this is to use the **Seek( )** method defined by **FileStream**. This method allows you to set the *file position indicator* (also called the *file pointer* or simply the *current position*) to any point within a file.

The method **Seek( )** is shown here:

long Seek(long *newPos*, SeekOrigin *origin*)

Here, *newPos* specifies the new position, in bytes, of the file pointer from the location specified by *origin.* The origin will be one of these values, which are defined by the **SeekOrigin** enumeration:

Value	Meaning
SeekOrigin.Begin	Seek from the beginning of the file.
SeekOrigin.Current	Seek from the current location.
SeekOrigin.End	Seek from the end of the file.

After a call to **Seek( )**, the next read or write operation will occur at the new file position. The new position is returned. If an error occurs while seeking, an **IOException** is thrown. If the underlying stream does not support position requests, a **NotSupportedException** is thrown. Other exceptions are possible.

Here is an example that demonstrates random access I/O. It writes the uppercase alphabet to a file and then reads it back in nonsequential order.

```
// Demonstrate random access.

using System;
using System.IO;

class RandomAccessDemo {
  static void Main() {
    FileStream f;
    char ch;
    try {
      f = new FileStream("random.dat", FileMode.Create);
    }
```

```
    catch(IOException exc) {
      Console.WriteLine("Cannot Open File");
      Console.WriteLine(exc.Message);
      return ;
    }

    // Write the alphabet.
    for(int i=0; i < 26; i++) {
      try {
        f.WriteByte((byte)('A'+i));
      }
      catch(IOException exc) {
        Console.WriteLine("Error Writing File");
        Console.WriteLine(exc.Message);
        f.Close();
        return ;
      }
    }

    try {
      // Now, read back specific values.
      f.Seek(0, SeekOrigin.Begin); // seek to first byte
      ch = (char) f.ReadByte();
      Console.WriteLine("First value is " + ch);

      f.Seek(1, SeekOrigin.Begin); // seek to second byte
      ch = (char) f.ReadByte();
      Console.WriteLine("Second value is " + ch);

      f.Seek(4, SeekOrigin.Begin); // seek to 5th byte
      ch = (char) f.ReadByte();
      Console.WriteLine("Fifth value is " + ch);

      Console.WriteLine();

      // Now, read every other value.
      Console.WriteLine("Here is every other value: ");
      for(int i=0; i < 26; i += 2) {
        f.Seek(i, SeekOrigin.Begin); // seek to ith character
        ch = (char) f.ReadByte();
        Console.Write(ch + " ");
      }
    }
    catch(IOException exc) {
      Console.WriteLine("Error Reading or Seeking");
      Console.WriteLine(exc.Message);
    }

    Console.WriteLine();
    f.Close();
  }
}
```

The output from the program is shown here:

```
First value is A
Second value is B
Fifth value is E

Here is every other value:
A C E G I K M O Q S U W Y
```

Although **Seek( )** offers the greatest flexibility, there is another way to set the current file position. You can use the **Position** property. As shown previously in Table 14-2, **Position** is a read/write property. Therefore, you can use it to obtain the current position or to set the current position. For example, here is the code sequence from the preceding program that reads the "random.dat" file, rewritten to use the **Position** property:

```
// Use Position rather than Seek() to set the current
// file position.
try {
  f.Position = 0;
  ch = (char) f.ReadByte();
  Console.WriteLine("First value is " + ch);

  f.Position = 1;
  ch = (char) f.ReadByte();
  Console.WriteLine("Second value is " + ch);

  f.Position = 4;
  ch = (char) f.ReadByte();
  Console.WriteLine("Fifth value is " + ch);

  Console.WriteLine();

  // Now, read every other value.
  Console.WriteLine("Here is every other value: ");
  for(int i=0; i < 26; i += 2) {
    f.Position = i; // seek to ith character
    ch = (char) f.ReadByte();
    Console.Write(ch + " ");
  }
}
catch(IOException exc) {
  Console.WriteLine("Error Reading or Seeking");
  Console.WriteLine(exc.Message);
}
```

Using MemoryStream

Sometimes it is useful to read input from or to write output to an array, rather than directly from or to a device. To do this, you will use **MemoryStream**. **MemoryStream** is an implementation of **Stream** that uses an array of bytes for input and/or output. **MemoryStream** defines several constructors. Here is the one we will use:

MemoryStream(byte[ ] *buf*)

Here, *buf* is an array of bytes that will be used for the source and/or target of I/O requests. The stream created by this constructor can be written or read, and supports **Seek( )**. When using this constructor, you must remember to make *buf* large enough to hold whatever output you will be directing to it.

Here is a program that demonstrates the use of **MemoryStream**:

```
// Demonstrate MemoryStream.

using System;
using System.IO;

class MemStrDemo {
  static void Main() {
    byte[] storage = new byte[255];

    // Create a memory-based stream.
    MemoryStream memstrm = new MemoryStream(storage);

    // Wrap memstrm in a reader and a writer.
    StreamWriter memwtr = new StreamWriter(memstrm);
    StreamReader memrdr = new StreamReader(memstrm);

    // Write to storage, through memwtr.
    for(int i=0; i < 10; i++)
       memwtr.WriteLine("byte [" + i + "]: " + i);

    // Put a period at the end.
    memwtr.WriteLine(".");

    memwtr.Flush();

    Console.WriteLine("Reading from storage directly: ");

    // Display contents of storage directly.
    foreach(char ch in storage) {
      if (ch == '.') break;
      Console.Write(ch);
    }

    Console.WriteLine("\nReading through memrdr: ");

    // Read from memstrm using the stream reader.
    memstrm.Seek(0, SeekOrigin.Begin); // reset file pointer

    string str = memrdr.ReadLine();
    while(str != null) {
      str = memrdr.ReadLine();
      if(str.CompareTo(".") == 0) break;
      Console.WriteLine(str);
    }
  }
}
```

The output from the program is shown here:

```
Reading from storage directly:
byte [0]: 0
byte [1]: 1
byte [2]: 2
byte [3]: 3
byte [4]: 4
byte [5]: 5
byte [6]: 6
byte [7]: 7
byte [8]: 8
byte [9]: 9

Reading through memrdr:
byte [1]: 1
byte [2]: 2
byte [3]: 3
byte [4]: 4
byte [5]: 5
byte [6]: 6
byte [7]: 7
byte [8]: 8
byte [9]: 9
```

In the program, an array of bytes called **storage** is created. This array is then used as the underlying storage for a **MemoryStream** called **memstrm**. From **memstrm** are created a **StreamReader** called **memrdr** and a **StreamWriter** called **memwtr**. Using **memwtr**, output is written to the memory-based stream. Notice that after the output has been written, **Flush( )** is called on **memwtr**. This is necessary to ensure that the contents of **memwtr**'s buffer are actually written to the underlying array. Next, the contents of the underlying byte array are displayed manually, using a **foreach** loop. Then, using **Seek( )**, the file pointer is reset to the start of the stream, and the memory stream is read using **memrdr**.

Memory-based streams are quite useful in programming. For example, you can construct complicated output in advance, storing it in the array until it is needed. This technique is especially useful when programming for a GUI environment, such as Windows. You can also redirect a standard stream to read from an array. This might be useful for feeding test information into a program, for example.

Using StringReader and StringWriter

For some applications, it might be easier to use a **string** rather than a **byte** array for the underlying storage when performing memory-based I/O operations. When this is the case, use **StringReader** and **StringWriter**. **StringReader** inherits **TextReader**, and **StringWriter** inherits **TextWriter**. Thus, these streams have access to methods defined by those two classes. For example, you can call **ReadLine( )** on a **StringReader** and **WriteLine( )** on a **StringWriter**.

The constructor for **StringReader** is shown here:

StringReader(string *str*)

Here, *str* is the string that will be read from.

StringWriter defines several constructors. Here is the one that we will use:

StringWriter( )

This constructor creates a writer that will put its output into a string. This string (in the form of a **StringBuilder**) is automatically created by **StringWriter**. You can obtain the contents of this string by calling **ToString( )**.

Here is an example that uses **StringReader** and **StringWriter**:

```
// Demonstrate StringReader and StringWriter.

using System;
using System.IO;

class StrRdrDemo {
  static void Main() {
    // Create a StringWriter.
    StringWriter strwtr = new StringWriter();

    // Write to StringWriter.
    for(int i=0; i < 10; i++)
      strwtr.WriteLine("This is i: " + i);

    // Create a StringReader.
    StringReader strrdr = new StringReader(strwtr.ToString());

    // Now, read from StringReader.
    string str = strrdr.ReadLine();
    while(str != null) {
      str = strrdr.ReadLine();
      Console.WriteLine(str);
    }
  }
}
```

The output is shown here:

```
This is i: 1
This is i: 2
This is i: 3
This is i: 4
This is i: 5
This is i: 6
This is i: 7
This is i: 8
This is i: 9
```

The program first creates a **StringWriter** called **strwtr** and outputs to it using **WriteLine( )**. Next, it creates a **StringReader** using the string contained in **strwtr**. This string is obtained by calling **ToString( )** on **strwtr**. Finally, the contents of this string are read using **ReadLine( )**.

Converting Numeric Strings to Their Internal Representation

Before leaving the topic of I/O, we will examine a technique useful when reading numeric strings. As you know, **WriteLine( )** provides a convenient way to output various types of data to the console, including numeric values of the built-in types, such as **int** and **double**. Thus, **WriteLine( )** automatically converts numeric values into their human-readable form. However, a parallel input method that reads and converts strings containing numeric values into their internal, binary format is not provided. For example, there is no version of **Read( )** that reads from the keyboard a string such as "100" and then automatically converts it into its corresponding binary value that can be stored in an **int** variable. Instead, there are other ways to accomplish this task. Perhaps the easiest is to use a method that is defined for all of the built-in numeric types: **Parse( )**.

Before we begin, it is necessary to state an important fact: All of C#'s built-in types, such as **int** and **double**, are actually just *aliases* (that is, other names) for structures defined by the .NET framework. In fact, the C# type and .NET structure type are indistinguishable. One is just another name for the other. Because C#'s value types are supported by structures, the value types have members defined for them.

For the numeric types, the .NET structure names and their C# keyword equivalents are shown here:

.NET Structure Name	C# Name
Decimal	decimal
Double	double
Single	float
Int16	short
Int32	int
Int64	long
UInt16	ushort
UInt32	uint
UInt64	ulong
Byte	byte
SByte	sbyte

These structures are defined inside the **System** namespace. Thus, the fully qualified name for **Int32** is **System.Int32**. These structures offer a wide array of methods that help fully integrate the value types into C#'s object hierarchy. As a side benefit, the numeric structures also define a static method called **Parse( )** that converts a numeric string into its corresponding binary equivalent.

There are several overloaded forms of **Parse( )**. The simplest version for each numeric structure is shown here. It performs the conversion using the default locale and numeric style. (Other versions let you perform locale-specific conversions and specify the numeric style.) Notice that each method returns a binary value that corresponds to the string.

Structure	Conversion Method
Decimal	static decimal Parse(string *str*)
Double	static double Parse(string *str*)
Single	static float Parse(string *str*)
Int64	static long Parse(string *str*)
Int32	static int Parse(string *str*)
Int16	static short Parse(string *str*)
UInt64	static ulong Parse(string *str*)
UInt32	static uint Parse(string *str*)
UInt16	static ushort Parse(string *str*)
Byte	static byte Parse(string *str*)
SByte	static sbyte Parse(string *str*)

The **Parse( )** methods will throw a **FormatException** if *str* does not contain a valid number as defined by the invoking type. **ArgumentNullException** is thrown if *str* is null, and **OverflowException** is thrown if the value in *str* exceeds the bounds of the invoking type.

The parsing methods give you an easy way to convert a numeric value, read as a string from the keyboard or a text file, into its proper internal format. For example, the following program averages a list of numbers entered by the user. It first asks the user for the number of values to be averaged. It then reads that number using **ReadLine( )** and uses **Int32.Parse( )** to convert the string into an integer. Next, it inputs the values, using **Double.Parse( )** to convert the strings into their **double** equivalents.

```
// This program averages a list of numbers entered by the user.

using System;
using System.IO;

class AvgNums {
  static void Main() {
    string str;
    int n;
    double sum = 0.0;
    double avg, t;

    Console.Write("How many numbers will you enter: ");
    str = Console.ReadLine();
    try {
      n = Int32.Parse(str);
    } catch(FormatException exc) {
      Console.WriteLine(exc.Message);
      return;
    } catch(OverflowException exc) {
      Console.WriteLine(exc.Message);
      return;
    }
```

```
    Console.WriteLine("Enter " + n + " values.");
    for(int i=0; i < n ; i++)  {
      Console.Write(": ");
      str = Console.ReadLine();
      try {
        t = Double.Parse(str);
      } catch(FormatException exc) {
        Console.WriteLine(exc.Message);
        t = 0.0;
      } catch(OverflowException exc) {
        Console.WriteLine(exc.Message);
        t = 0;
      }
      sum += t;
    }
    avg = sum / n;
    Console.WriteLine("Average is " + avg);
  }
}
```

Here is a sample run:

```
How many numbers will you enter: 5
Enter 5 values.
: 1.1
: 2.2
: 3.3
: 4.4
: 5.5
Average is 3.3
```

One other point: You must use the right parsing method for the type of value you are trying to convert. For example, trying to use **Int32.Parse( )** on a string that contains a floating-point value will not produce the desired result.

As explained, **Parse( )** will throw an exception on failure. You can avoid generating an exception when converting numeric strings by using the **TryParse( )** method, which is defined for all of the numeric structures. Here is an example. It shows one version of **TryParse( )** as defined by **Int32**.

static bool TryParse(string *str*, out int *result*)

The numeric string is passed in *str*. The result is returned in *result*. It performs the conversion using the default locale and numeric style. (A second version of **TryParse( )** is available that lets you specify the numeric style and locale.) If the conversion fails, such as when *str* does not contain a numeric string in the proper form, **TryParse( )** returns false. Otherwise, it returns true. Therefore, you must check the return value to confirm that a successful conversion has occurred.

CHAPTER 15

Delegates, Events, and Lambda Expressions

This chapter examines three innovative C# features: delegates, events, and lambda expressions. A *delegate* provides a way to encapsulate a method. An *event* is a notification that some action has occurred. Delegates and events are related because an event is built upon a delegate. Both expand the set of programming tasks to which C# can be applied. The *lambda expression* is a new syntactic feature provided by C# 3.0. It offers a streamlined, yet powerful way to define what is, essentially, a unit of executable code. Lambda expressions are often used when working with delegates and events because a delegate can refer to a lambda expression. (Lambda expressions are also very important to LINQ, which is described in Chapter 19.) Also examined are anonymous methods, covariance, contravariance, and method group conversions.

Delegates

Let's begin by defining the term *delegate.* In straightforward language, a delegate is an object that can refer to a method. Therefore, when you create a delegate, you are creating an object that can hold a reference to a method. Furthermore, the method can be called through this reference. In other words, a delegate can invoke the method to which it refers. As you will see, this is a very powerful concept.

It is important to understand that the same delegate can be used to call different methods during the runtime of a program by simply changing the method to which the delegate refers. Thus, the method that will be invoked by a delegate is not determined at compile time, but rather at runtime. This is the principal advantage of a delegate.

NOTE *If you are familiar with C/C++, then it will help to know that a delegate in C# is similar to a function pointer in C/C++.*

A delegate type is declared using the keyword **delegate**. The general form of a delegate declaration is shown here:

delegate *ret-type name*(*parameter-list*);

Here, *ret-type* is the type of value returned by the methods that the delegate will be calling. The name of the delegate is specified by *name*. The parameters required by the methods called through the delegate are specified in the *parameter-list*. Once created, a delegate instance can refer to and call methods whose return type and parameter list match those specified by the delegate declaration.

A key point to understand is that a delegate can be used to call *any* method that agrees with its signature and return type. Furthermore, the method can be either an instance method associated with an object or a **static** method associated with a class. All that matters is that the return type and signature of the method agree with those of the delegate.

To see delegates in action, let's begin with the simple example shown here:

```
// A simple delegate example.

using System;

// Declare a delegate type.
delegate string StrMod(string str);

class DelegateTest {
  // Replaces spaces with hyphens.
  static string ReplaceSpaces(string s) {
    Console.WriteLine("Replacing spaces with hyphens.");
    return s.Replace(' ', '-');
  }

  // Remove spaces.
  static string RemoveSpaces(string s) {
    string temp = "";
    int i;

    Console.WriteLine("Removing spaces.");
    for(i=0; i < s.Length; i++)
      if(s[i] != ' ') temp += s[i];

    return temp;
  }

  // Reverse a string.
  static string Reverse(string s) {
    string temp = "";
    int i, j;

    Console.WriteLine("Reversing string.");
    for(j=0, i=s.Length-1; i >= 0; i--, j++)
      temp += s[i];

    return temp;
  }

  static void Main() {
    // Construct a delegate.
    StrMod strOp = new StrMod(ReplaceSpaces);
    string str;
```

```
    // Call methods through the delegate.
    str = strOp("This is a test.");
    Console.WriteLine("Resulting string: " + str);
    Console.WriteLine();

    strOp = new StrMod(RemoveSpaces);
    str = strOp("This is a test.");
    Console.WriteLine("Resulting string: " + str);
    Console.WriteLine();

    strOp = new StrMod(Reverse);
    str = strOp("This is a test.");
    Console.WriteLine("Resulting string: " + str);
  }
}
```

The output from the program is shown here:

```
Replacing spaces with hyphens.
Resulting string: This-is-a-test.

Removing spaces.
Resulting string: Thisisatest.

Reversing string.
Resulting string: .tset a si sihT
```

Let's examine this program closely. The program declares a delegate type called **StrMod**, shown here:

```
delegate string StrMod(string str);
```

Notice that **StrMod** takes one **string** parameter and returns a **string**.

Next, in **DelegateTest**, three **static** methods are declared, each with a single parameter of type **string** and a return type of **string**. Thus, they match the **StrMod** delegate. These methods perform some type of string modification. Notice that **ReplaceSpaces( )** uses one of **string**'s methods, called **Replace( )**, to replace spaces with hyphens.

In **Main( )**, a **StrMod** reference called **strOp** is created and assigned a reference to **ReplaceSpaces( )**. Pay close attention to this line:

```
StrMod strOp = new StrMod(ReplaceSpaces);
```

Notice how the method **ReplaceSpaces( )** is passed as a parameter. Only its name is used; no parameters are specified. This can be generalized. When instantiating a delegate, you specify only the name of the method to which you want the delegate to refer. Of course, the method's signature must match that of the delegate's declaration. If it doesn't, a compile-time error will result.

Next, **ReplaceSpaces( )** is called through the delegate instance **strOp**, as shown here:

```
str = strOp("This is a test.");
```

Because **strOp** refers to **ReplaceSpaces( )**, **ReplaceSpaces( )** is invoked.

Next, **strOp** is assigned a reference to **RemoveSpaces( )**, and then **strOp** is called again. This time, **RemoveSpaces( )** is invoked.

Finally, **strOp** is assigned a reference to **Reverse( )** and **strOp** is called. This results in **Reverse( )** being called.

The key point of the example is that the invocation of **strOp** results in a call to the method referred to by **strOp** at the time at which the invocation occurs. Thus, the method to call is resolved at runtime, not compile time.

Delegate Method Group Conversion

Beginning with version 2.0, C# has included an option that significantly simplifies the syntax that assigns a method to a delegate. This feature is called *method group conversion*, and it allows you to simply assign the name of a method to a delegate, without using **new** or explicitly invoking the delegate's constructor.

For example, here is the **Main( )** method of the preceding program rewritten to use method group conversions:

```
static void Main() {
  // Construct a delegate using method group conversion.
  StrMod strOp = ReplaceSpaces; // use method group conversion
  string str;

  // Call methods through the delegate.
  str = strOp("This is a test.");
  Console.WriteLine("Resulting string: " + str);
  Console.WriteLine();

  strOp = RemoveSpaces; // use method group conversion
  str = strOp("This is a test.");
  Console.WriteLine("Resulting string: " + str);
  Console.WriteLine();

  strOp = Reverse; // use method group conversion
  str = strOp("This is a test.");
  Console.WriteLine("Resulting string: " + str);
}
```

Pay special attention to the way that **strOp** is created and assigned the method **ReplaceSpaces** in this line:

```
StrMod strOp = ReplaceSpaces; // use method group conversion
```

The name of the method is assigned directly to **strOp**. C# automatically provides a conversion from the method to the delegate type. This syntax can be generalized to any situation in which a method is assigned to (or converted to) a delegate type.

Because the method group conversion syntax is simpler than the old approach, it is used throughout the remainder of this book.

Using Instance Methods as Delegates

Although the preceding example used **static** methods, a delegate can also refer to instance methods. It must do so, however, through an object reference. For example, here is a rewrite of the previous example, which encapsulates the string operations inside a class called

StringOps. Notice that the method group conversion syntax can also be applied in this situation.

```
// Delegates can refer to instance methods, too.

using System;

// Declare a delegate type.
delegate string StrMod(string str);

class StringOps {
  // Replaces spaces with hyphens.
  public string ReplaceSpaces(string s) {
    Console.WriteLine("Replacing spaces with hyphens.");
    return s.Replace(' ', '-');
  }

  // Remove spaces.
  public string RemoveSpaces(string s) {
    string temp = "";
    int i;

    Console.WriteLine("Removing spaces.");
    for(i=0; i < s.Length; i++)
      if(s[i] != ' ') temp += s[i];

    return temp;
  }

  // Reverse a string.
  public string Reverse(string s) {
    string temp = "";
    int i, j;

    Console.WriteLine("Reversing string.");
    for(j=0, i=s.Length-1; i >= 0; i--, j++)
      temp += s[i];

    return temp;
  }
}

class DelegateTest {
  static void Main() {
    StringOps so = new StringOps(); // create an instance of StringOps

    // Initialize a delegate.
    StrMod strOp = so.ReplaceSpaces;
    string str;

    // Call methods through delegates.
    str = strOp("This is a test.");
    Console.WriteLine("Resulting string: " + str);
    Console.WriteLine();
```

```
    strOp = so.RemoveSpaces;
    str = strOp("This is a test.");
    Console.WriteLine("Resulting string: " + str);
    Console.WriteLine();

    strOp = so.Reverse;
    str = strOp("This is a test.");
    Console.WriteLine("Resulting string: " + str);
  }
}
```

This program produces the same output as the first, but in this case, the delegate refers to methods on an instance of **StringOps**.

Multicasting

One of the most exciting features of a delegate is its support for *multicasting*. In simple terms, multicasting is the ability to create an *invocation list*, or chain, of methods that will be automatically called when a delegate is invoked. Such a chain is very easy to create. Simply instantiate a delegate, and then use the **+** or **+=** operator to add methods to the chain. To remove a method, use **–** or **– =**. If the delegate returns a value, then the value returned by the last method in the list becomes the return value of the entire delegate invocation. Thus, a delegate that makes use of multicasting will often have a **void** return type.

Here is an example of multicasting. Notice that it reworks the preceding examples by changing the string manipulation method's return type to **void** and using a **ref** parameter to return the altered string to the caller. This makes the methods more appropriate for multicasting.

```
// Demonstrate multicasting.

using System;

// Declare a delegate type.
delegate void StrMod(ref string str);

class MultiCastDemo {
  // Replaces spaces with hyphens.
  static void ReplaceSpaces(ref string s) {
    Console.WriteLine("Replacing spaces with hyphens.");
    s = s.Replace(' ', '-');
  }

  // Remove spaces.
  static void RemoveSpaces(ref string s) {
    string temp = "";
    int i;

    Console.WriteLine("Removing spaces.");
    for(i=0; i < s.Length; i++)
      if(s[i] != ' ') temp += s[i];

    s = temp;
  }
```

```
  // Reverse a string.
  static void Reverse(ref string s) {
    string temp = "";
    int i, j;

    Console.WriteLine("Reversing string.");
    for(j=0, i=s.Length-1; i >= 0; i--, j++)
      temp += s[i];

    s = temp;
  }

  static void Main() {
    // Construct delegates.
    StrMod strOp;
    StrMod replaceSp = ReplaceSpaces;
    StrMod removeSp = RemoveSpaces;
    StrMod reverseStr = Reverse;
    string str = "This is a test";

    // Set up multicast.
    strOp = replaceSp;
    strOp += reverseStr;

    // Call multicast.
    strOp(ref str);
    Console.WriteLine("Resulting string: " + str);
    Console.WriteLine();

    // Remove replace and add remove.
    strOp -= replaceSp;
    strOp += removeSp;

    str = "This is a test."; // reset string

    // Call multicast.
    strOp(ref str);
    Console.WriteLine("Resulting string: " + str);
    Console.WriteLine();
  }
}
```

Here is the output:

```
Replacing spaces with hyphens.
Reversing string.
Resulting string: tset-a-si-sihT

Reversing string.
Removing spaces.
Resulting string: .tsetasisihT
```

In **Main( )**, four delegate instances are created. One, **strOp**, is null. The other three refer to specific string modification methods. Next, a multicast is created that calls **RemoveSpaces( )** and **Reverse( )**. This is accomplished via the following lines:

```
strOp = replaceSp;
strOp += reverseStr;
```

First, **strOp** is assigned **replaceSp**. Next, using **+=**, **reverseStr** is added. When **strOp** is invoked, both methods are invoked, replacing spaces with hyphens and reversing the string, as the output illustrates.

Next, **replaceSp** is removed from the chain, using this line:

```
strOp -= replaceSp;
```

and **removeSP** is added using this line:

```
strOp += removeSp;
```

Then, **strOp** is again invoked. This time, spaces are removed and the string is reversed.

Delegate chains are a powerful mechanism because they allow you to define a set of methods that can be executed as a unit. This can increase the structure of some types of code. Also, as you will soon see, delegate chains have a special value to events.

Covariance and Contravariance

There are two features that add flexibility to delegates: *covariance* and *contravariance*. Normally, the method that you pass to a delegate must have the same return type and signature as the delegate. However, covariance and contravariance relax this rule slightly, as it pertains to derived types. Covariance enables a method to be assigned to a delegate when the method's return type is a class derived from the class specified by the return type of the delegate. Contravariance enables a method to be assigned to a delegate when a method's parameter type is a base class of the class specified by the delegate's declaration.

Here is an example that illustrates both covariance and contravariance:

```
// Demonstrate covariance and contravariance.

using System;

class X {
  public int Val;
}

// Y is derived from X.
class Y : X { }

// This delegate returns X and takes a Y argument.
delegate X ChangeIt(Y obj);

class CoContraVariance {

  // This method returns X and has an X parameter.
```

```
  static X IncrA(X obj) {
    X temp = new X();
    temp.Val = obj.Val + 1;
    return temp;
  }

  // This method returns Y and has a Y parameter.
  static Y IncrB(Y obj) {
    Y temp = new Y();
    temp.Val = obj.Val + 1;
    return temp;
  }

  static void Main() {
    Y Yob = new Y();

    // In this case, the parameter to IncrA
    // is X and the parameter to ChangeIt is Y.
    // Because of contravariance, the following
    // line is OK.
    ChangeIt change = IncrA;

    X Xob = change(Yob);

    Console.WriteLine("Xob: " + Xob.Val);

    // In the next case, the return type of
    // IncrB is Y and the return type of
    // ChangeIt is X. Because of covariance,
    // the following line is OK.
    change = IncrB;

    Yob = (Y) change(Yob);

    Console.WriteLine("Yob: " + Yob.Val);
  }
}
```

The output from the program is shown here:

```
Xob: 1
Yob: 1
```

In the program, notice that class **Y** is derived from class **X**. Next, notice that the delegate **ChangeIt( )** is declared like this:

```
delegate X ChangeIt(Y obj);
```

ChangeIt( ) returns **X** and has a **Y** parameter. Next, notice that the methods **IncrA( )** and **IncrB( )** are declared as shown here:

```
static X IncrA(X obj)
static Y IncrB(Y obj)
```

The **IncrA( )** method has an **X** parameter and returns **X**. The **IncrB( )** method has a **Y** parameter and returns **Y**. Given covariance and contravariance, either of these methods can be passed to **ChangeIt**, as the program illustrates.

Therefore, this line

```
ChangeIt change = IncrA;
```

uses contravariance to enable **IncrA( )** to be passed to the delegate because **IncrA( )** has an **X** parameter, but the delegate has a **Y** parameter. This works because, with contravariance, if the parameter type of the method passed to a delegate is a base class of the parameter type used by the delegate, then the method and the delegate are compatible.

The next line is also legal, but this time it is because of covariance:

```
change = IncrB;
```

In this case, the return type of **IncrB( )** is **Y**, but the return type of **ChangeIt( )** is **X**. However, because the return type of the method is a class derived from the return type of the delegate, the two are compatible.

System.Delegate

All delegates are classes that are implicitly derived from **System.Delegate**. You don't normally need to use its members directly, and this book makes no explicit use of **System.Delegate**. However, its members may be useful in certain specialized situations.

Why Delegates

Although the preceding examples show the "how" behind delegates, they don't really illustrate the "why." In general, delegates are useful for two main reasons. First, as shown later in this chapter, delegates support events. Second, delegates give your program a way to execute methods at runtime without having to know precisely what those methods are at compile time. This ability is quite useful when you want to create a framework that allows components to be plugged in. For example, imagine a drawing program (a bit like the standard Windows Paint accessory). Using a delegate, you could allow the user to plug in special color filters or image analyzers. Furthermore, the user could create a sequence of these filters or analyzers. Such a scheme could be easily handled using a delegate.

Anonymous Functions

You will often find that the method referred to by a delegate is used only for that purpose. In other words, the only reason for the method is so it can be invoked via a delegate. The method is never called on its own. In such a case, you can avoid the need to create a separate method by using an *anonymous function.* An anonymous function is, essentially, an unnamed block of code that is passed to a delegate constructor. One advantage to using an anonymous function is simplicity. There is no need to declare a separate method whose only purpose is to be passed to a delegate.

Beginning with version 3.0, C# defines two types of anonymous functions: *anonymous methods* and *lambda expressions.* The anonymous method was added by C# 2.0. The lambda expression was added by C# 3.0. In general, the lambda expression improves on the concept

of the anonymous method and is now the preferred approach to creating an anonymous function. However, anonymous methods are widely used in existing C# code. Therefore, they are still an important part of C#. Furthermore, anonymous methods are the precursor to lambda expressions and a clear understanding of anonymous methods makes it easier to understand aspects of the lambda expression. Also, there is a narrow set of cases in which an anonymous method can be used, but a lambda expression cannot. Therefore, both anonymous methods and lambda expressions are described in this chapter.

Anonymous Methods

An anonymous method is one way to create an unnamed block of code that is associated with a specific delegate instance. An anonymous method is created by following the keyword **delegate** with a block of code. To see how this is done, let's begin with a simple example. The following program uses an anonymous method that counts from 0 to 5.

```
// Demonstrate an anonymous method.

using System;

// Declare a delegate type.
delegate void CountIt();

class AnonMethDemo {

  static void Main() {

    // Here, the code for counting is passed
    // as an anonymous method.
    CountIt count = delegate {
        // This is the block of code passed to the delegate.
      for(int i=0; i <= 5; i++)
        Console.WriteLine(i);
    }; // notice the semicolon

    count();
  }
}
```

This program first declares a delegate type called **CountIt** that has no parameters and returns **void**. Inside **Main( )**, a **CountIt** instance called **count** is created, and it is passed the block of code that follows the **delegate** keyword. This block of code is the anonymous method that will be executed when **count** is called. Notice that the block of code is followed by a semicolon, which terminates the declaration statement. The output from the program is shown here:

```
0
1
2
3
4
5
```

Pass Arguments to an Anonymous Method

It is possible to pass one or more arguments to an anonymous method. To do so, follow the **delegate** keyword with a parenthesized parameter list. Then, pass the argument(s) to the delegate instance when it is called. For example, here is the preceding program rewritten so that the ending value for the count is passed:

```
// Demonstrate an anonymous method that takes an argument.

using System;

// Notice that CountIt now has a parameter.
delegate void CountIt(int end);

class AnonMethDemo2 {

  static void Main() {

    // Here, the ending value for the count
    // is passed to the anonymous method.
    CountIt count = delegate (int end) {
      for(int i=0; i <= end; i++)
        Console.WriteLine(i);
    };

    count(3);
    Console.WriteLine();
    count(5);
  }
}
```

In this version, **CountIt** now takes an integer argument. Notice how the parameter list is specified after the **delegate** keyword when the anonymous method is created. The code inside the anonymous method has access to the parameter **end** in just the same way it would if a named method were being created. The output from this program is shown next:

```
0
1
2
3

0
1
2
3
4
5
```

Return a Value from an Anonymous Method

An anonymous method can return a value. The value is returned by use of the **return** statement, which works the same in an anonymous method as it does in a named method. As you would expect, the type of the return value must be compatible with the return type

specified by the delegate. For example, here the code that performs the count also computes the summation of the count and returns the result:

```
// Demonstrate an anonymous method that returns a value.

using System;

// This delegate returns a value.
delegate int CountIt(int end);

class AnonMethDemo3 {

  static void Main() {
    int result;

    // Here, the ending value for the count
    // is passed to the anonymous method.
    // A summation of the count is returned.
    CountIt count = delegate (int end) {
      int sum = 0;

      for(int i=0; i <= end; i++) {
        Console.WriteLine(i);
        sum += i;
      }
      return sum; // return a value from an anonymous method
    };

    result = count(3);
    Console.WriteLine("Summation of 3 is " + result);
    Console.WriteLine();

    result = count(5);
    Console.WriteLine("Summation of 5 is " + result);
  }
}
```

In this version, the value of **sum** is returned by the code block that is associated with the **count** delegate instance. Notice that the **return** statement is used in an anonymous method in just the same way that it is used in a named method. The output is shown here:

```
0
1
2
3
Summation of 3 is 6

0
1
2
3
4
5
Summation of 5 is 15
```

Use Outer Variables with Anonymous Methods

A local variable or parameter whose scope includes an anonymous method is called an *outer variable.* An anonymous method has access to and can use these outer variables. When an outer variable is used by an anonymous method, that variable is said to be *captured.* A captured variable will stay in existence at least until the delegate that captured it is subject to garbage collection. Thus, even though a local variable will normally cease to exist when its block is exited, if that local variable is being used by an anonymous method, then that variable will stay in existence at least until the delegate referring to that method is destroyed.

The capturing of a local variable can lead to unexpected results. For example, consider this version of the counting program. As in the previous version, the summation of the count is computed. However, in this version, a **CountIt** object is constructed and returned by a static method called **Counter( )**. This object uses the variable **sum**, which is declared in the enclosing scope provided by **Counter( )**, rather than in the anonymous method, itself. Thus, **sum** is captured by the anonymous method. Inside **Main( )**, **Counter( )** is called to obtain a **CountIt** object. Thus, **sum** will not be destroyed until the program finishes.

```
// Demonstrate a captured variable.

using System;

// This delegate returns int and takes an int argument.
delegate int CountIt(int end);

class VarCapture {

  static CountIt Counter() {
    int sum = 0;

    // Here, a summation of the count is stored
    // in the captured variable sum.
    CountIt ctObj = delegate (int end) {
      for(int i=0; i <= end; i++) {
        Console.WriteLine(i);
        sum += i;
      }
      return sum;
    };
    return ctObj;
  }

  static void Main() {
    // Get a counter.
    CountIt count = Counter();

    int result;

    result = count(3);
    Console.WriteLine("Summation of 3 is " + result);
    Console.WriteLine();

    result = count(5);
    Console.WriteLine("Summation of 5 is " + result);
```

```
  }
}
```

The output is shown here. Pay special attention to the summation value.

```
0
1
2
3
Summation of 3 is 6

0
1
2
3
4
5
Summation of 5 is 21
```

As you can see, the count still proceeds normally. However, notice the summation value for 5. It shows 21 instead of 15! The reason for this is that **sum** is captured by **ctObj** when it is created by the **Counter( )** method. This means it remains in existence until **count** is subject to garbage collection at the end of the program. Thus, its value is not destroyed when **Counter( )** returns or with each call to the anonymous method when **count** is called in **Main( )**.

Although captured variables can result in rather counterintuitive situations, such as the one just shown, it makes sense if you think about it a bit. The key point is that when an anonymous method captures a variable, that variable cannot go out of existence until the delegate that captures it is no longer being used. If this were not the case, then the captured variable could be undefined when it is needed by the delegate.

Lambda Expressions

Although anonymous methods are a valuable feature, they have been largely superceded by a better approach: the *lambda expression.* It is not an overstatement to say that the lambda expression is one of the two most important features added by C# 3.0 (the other being LINQ). Based on an entirely new syntactic element, the lambda expression provides a powerful alternative to the anonymous method. Although a principal use of lambda expressions is found when working with LINQ (see Chapter 19), they are also applicable to (and commonly used with) delegates and events. This use of lambda expressions is described here.

A lambda expression is the second way that an anonymous function can be created. (The other type of anonymous function is the anonymous method, described in the preceding section.) Thus, a lambda expression can be assigned to a delegate. Because a lambda expression is more streamlined than the equivalent anonymous method, lambda expressions are now the recommended approach in almost all cases.

The Lambda Operator

All lambda expressions use the new *lambda operator,* which is **=>**. This operator divides a lambda expression into two parts. On the left the input parameter (or parameters) is specified. On the right is the lambda body. The **=>** operator is sometimes verbalized as "goes to" or "becomes."

C# supports two types of lambda expressions, and it is the lambda body that determines what type is being created. If the lambda body consists of a single expression, then an *expression lambda* is being created. In this case, the body is free-standing—it is not enclosed between braces. If the lambda body consists of a block of statements enclosed by braces, then a *statement lambda* is being created. A statement lambda can contain multiple statements and include such things as loops, method calls, and **if** statements. The following sections describe both kinds of lambdas.

Expression Lambdas

In an expression lambda, the expression on the right side of the **=>** acts on the parameter (or parameters) specified by the left side. The result of the expression becomes the result of the lambda operator and is returned.

Here is the general form of an expression lambda that takes only one parameter:

param => expr

When more than one parameter is required, then the following form is used:

(*param-list*) => *expr*

Therefore, when two or more parameters are needed, they must be enclosed by parentheses. If no parameters are needed, then empty parentheses must be used.

Here is a simple expression lambda:

count => count + 2

Here **count** is the parameter that is acted on by the expression **count + 2**. Thus, the result is the value of **count** increased by two. Here is another example:

n => n % 2 == 0

In this case, this expression returns true if **n** is even and false if it is odd.

To use a lambda expression involves two steps. First, declare a delegate type that is compatible with the lambda expression. Second, declare an instance of the delegate, assigning to it the lambda expression. Once this has been done, the lambda expression can be executed by calling the delegate instance. The result of the lambda expression becomes the return value.

The following program shows how to put the two expression lambdas just shown into action. It declares two delegate types. The first, called **Incr**, takes an **int** argument and returns an **int** result. The second, called **IsEven**, takes an **int** argument and returns a **bool** result. It then assigns the lambda expressions to instances of those delegates. Finally, it executes the lambda expressions through the delegate instances.

```
// Use two simple lambda expressions.

using System;

// Declare a delegate that takes an int argument
// and returns an int result.
delegate int Incr(int v);

// Declare a delegate that takes an int argument
// and returns a bool result.
```

```
delegate bool IsEven(int v);

class SimpleLambdaDemo {

  static void Main() {

    // Create an Incr delegate instance that refers to
    // a lambda expression that increases its parameter by 2.
    Incr incr = count => count + 2;

    // Now, use the incr lambda expression.
    Console.WriteLine("Use incr lambda expression: ");
    int x = -10;
    while(x <= 0) {
      Console.Write(x + " ");
      x = incr(x); // increase x by 2
    }

    Console.WriteLine("\n");

    // Create an IsEven delegate instance that refers to
    // a lambda expression that returns true if its parameter
    // is even and false otherwise.
    IsEven isEven = n => n % 2 == 0;

    // Now, use the isEven lambda expression.
    Console.WriteLine("Use isEven lambda expression: ");
    for(int i=1; i <= 10; i++)
      if(isEven(i)) Console.WriteLine(i + " is even.");

  }
}
```

The output is shown here:

```
Use incr lambda expression:
-10 -8 -6 -4 -2 0

Use isEven lambda expression:
2 is even.
4 is even.
6 is even.
8 is even.
10 is even.
```

In the program, pay special attention to these declarations:

```
Incr incr = count => count + 2;
IsEven isEven = n => n % 2 == 0;
```

The first assigns to **incr** a lambda expression that returns the result of increasing the value passed to **count** by 2. This expression can be assigned to an **Incr** delegate because it is compatible with **Incr**'s declaration. The argument used in the call to **incr** is passed to **count**. The result is returned. The second declaration assigns to **isEven** an expression that returns

true if the argument is even and false otherwise. Thus, it is compatible with the **IsEven** delegate declaration.

At this point, you might be wondering how the compiler knows the type of the data used in a lambda expression. For example, in the lambda expression assigned to **incr**, how does the compiler know that **count** is an **int**? The answer is that the compiler infers the type of the parameter and the expression's result type from the delegate type. Thus, the lambda parameters and return value must be compatible with the parameter type(s) and return type of the delegate.

Although type inference is quite useful, in some cases, you might need to explicitly specify the type of a lambda parameter. To do so, simply include the type name. For example, here is another way to declare the **incr** delegate instance:

```
Incr incr = (int count) => count + 2;
```

Notice now that **count** is explicitly declared as an **int**. Also notice the use of parentheses. They are now necessary. (Parentheses can be omitted only when exactly one parameter is specified and no type specifier is used.)

Although the preceding two lambda expressions each used one parameter, lambda expressions can use any number, including zero. When using more than one parameter you *must* enclose them within parentheses. Here is an example that uses a lambda expression to determine if a value is within a specified range:

```
(low, high, val) => val >= low && val <= high;
```

Here is a delegate type that is compatible with this lambda expression:

```
delegate bool InRange(int lower, int upper, int v);
```

Thus, you could create an **InRange** delegate instance like this:

```
InRange rangeOK = (low, high, val) => val >= low && val <= high;
```

After doing so, the lambda expression can be executed as shown here:

```
if(rangeOK(1, 5, 3)) Console.WriteLine("3 is within 1 to 5.");
```

One other point: Lambda expressions can use outer variables in the same way as anonymous methods, and they are captured in the same way.

Statement Lambdas

As mentioned, there are two basic flavors of the lambda expression. The first is the expression lambda, which was discussed in the preceding section. As explained, the body of an expression lambda consists solely of a single expression. The second type of lambda expression is the *statement lambda.* A statement lambda expands the types of operations that can be handled within a lambda expression because it allows the body of lambda to contain multiple statements. For example, using a statement lambda you can use loops, **if** statements, declare variables, and so on. A statement lambda is easy to create. Simply enclose the body within braces. Aside from allowing multiple statements, it works much like the expression lambdas just discussed.

Here is an example that uses a statement lambda to compute and return the factorial of an **int** value:

```
// Demonstrate a statement lambda.
using System;

// IntOp takes one int argument and returns an int result.
delegate int IntOp(int end);

class StatementLambdaDemo {

  static void Main() {

    // A statement lambda that returns the factorial
    // of the value it is passed.
    IntOp fact = n => {
                        int r = 1;
                        for(int i=1; i <= n; i++)
                          r = i * r;
                        return r;
                      };

    Console.WriteLine("The factorial of 3 is " + fact(3));
    Console.WriteLine("The factorial of 5 is " + fact(5));
  }
}
```

The output is shown here:

```
The factorial of 3 is 6
The factorial of 5 is 120
```

In the program, notice that the statement lambda declares a variable called **r**, uses a **for** loop, and has a **return** statement. These are legal inside a statement lambda. In essence, a statement lambda closely parallels an anonymous method. Therefore, many anonymous methods will be converted to statement lambdas when updating legacy code. (As mentioned, as of C# 3.0, lambda expressions are the preferred way of creating anonymous functions.) One other point: When a **return** statement occurs within a lambda expression, it simply causes a return from the lambda. It does not cause the enclosing method to return.

Before concluding, it is worthwhile to see another example that shows the statement lambda in action. The following program reworks the first delegate example in this chapter so it uses statement lambdas (rather than standalone methods) to accomplish various string modifications:

```
// The first delegate example rewritten to use
// statement lambdas.

using System;

// Declare a delegate type.
delegate string StrMod(string s);
```

```
class UseStatementLambdas {

  static void Main() {
    // Create delegates that refer to lambda expressions
    // that perform various string modifications.

    // Replaces spaces with hyphens.
    StrMod ReplaceSpaces = s => {
             Console.WriteLine("Replacing spaces with hyphens.");
             return s.Replace(' ', '-');
           };

    // Remove spaces.
    StrMod RemoveSpaces = s => {
             string temp = "";
             int i;

             Console.WriteLine("Removing spaces.");
             for(i=0; i < s.Length; i++)
               if(s[i] != ' ') temp += s[i];

             return temp;
           };


    // Reverse a string.
    StrMod Reverse = s => {
             string temp = "";
             int i, j;

             Console.WriteLine("Reversing string.");
             for(j=0, i=s.Length-1; i >= 0; i--, j++)
             temp += s[i];

             return temp;
           };

    string str;

    // Call methods through the delegate.
    StrMod strOp = ReplaceSpaces;
    str = strOp("This is a test.");
    Console.WriteLine("Resulting string: " + str);
    Console.WriteLine();

    strOp = RemoveSpaces;
    str = strOp("This is a test.");
    Console.WriteLine("Resulting string: " + str);
    Console.WriteLine();

    strOp = Reverse;
    str = strOp("This is a test.");
```

```
    Console.WriteLine("Resulting string: " + str);
  }
}
```

The output, which is the same as the original version, is shown here:

```
Replacing spaces with hyphens.
Resulting string: This-is-a-test.

Removing spaces.
Resulting string: Thisisatest.

Reversing string.
Resulting string: .tset a si sihT
```

Events

Another important C# feature is built upon the foundation of delegates: the *event*. An event is, essentially, an automatic notification that some action has occurred. Events work like this: An object that has an interest in an event registers an event handler for that event. When the event occurs, all registered handlers are called. Event handlers are represented by delegates.

Events are members of a class and are declared using the **event** keyword. Its most commonly used form is shown here:

event *event-delegate event-name*;

Here, *event-delegate* is the name of the delegate used to support the event, and *event-name* is the name of the specific event object being declared.

Let's begin with a very simple example:

```
// A very simple event demonstration.

using System;

// Declare a delegate type for an event.
delegate void MyEventHandler();

// Declare a class that contains an event.
class MyEvent {
  public event MyEventHandler SomeEvent;

  // This is called to fire the event.
  public void OnSomeEvent() {
    if(SomeEvent != null)
      SomeEvent();
  }
}

class EventDemo {
  // An event handler.
  static void Handler() {
    Console.WriteLine("Event occurred");
  }
```

```
  static void Main() {
    MyEvent evt = new MyEvent();

    // Add Handler() to the event list.
    evt.SomeEvent += Handler;

    // Fire the event.
    evt.OnSomeEvent();
  }
}
```

This program displays the following output:

```
Event occurred
```

Although simple, this program contains all the elements essential to proper event handling. Let's look at it carefully. The program begins by declaring a delegate type for the event handler, as shown here:

```
delegate void MyEventHandler();
```

All events are activated through a delegate. Thus, the event delegate type defines the return type and signature for the event. In this case, there are no parameters, but event parameters are allowed.

Next, an event class, called **MyEvent**, is created. Inside the class, an event called **SomeEvent** is declared, using this line:

```
public event MyEventHandler SomeEvent;
```

Notice the syntax. The keyword **event** tells the compiler that an event is being declared.

Also declared inside **MyEvent** is the method **OnSomeEvent( )**, which is the method a program will call to signal (or "fire") an event. (That is, this is the method called when the event occurs.) It calls an event handler through the **SomeEvent** delegate, as shown here:

```
if(SomeEvent != null)
  SomeEvent();
```

Notice that a handler is called if and only if **SomeEvent** is not **null**. Since other parts of your program must register an interest in an event in order to receive event notifications, it is possible that **OnSomeEvent( )** could be called before any event handler has been registered. To prevent calling on a **null** reference, the event delegate must be tested to ensure that it is not **null**.

Inside **EventDemo**, an event handler called **Handler( )** is created. In this simple example, the event handler simply displays a message, but other handlers could perform more meaningful actions. In **Main( )**, a **MyEvent** object is created, and **Handler( )** is registered as a handler for this event, by adding it as shown here:

```
MyEvent evt = new MyEvent();

// Add Handler() to the event list.
evt.SomeEvent += Handler;
```

Notice that the handler is added using the += operator. Events support only += and –=. In this case, **Handler( )** is a **static** method, but event handlers can also be instance methods.

Finally, the event is fired as shown here:

```
// Fire the event.
evt.OnSomeEvent();
```

Calling **OnSomeEvent( )** causes all registered event handlers to be called. In this case, there is only one registered handler, but there could be more, as the next section explains.

A Multicast Event Example

Like delegates, events can be multicast. This enables multiple objects to respond to an event notification. Here is an event multicast example:

```
// An event multicast demonstration.

using System;

// Declare a delegate type for an event.
delegate void MyEventHandler();

// Declare a class that contains an event.
class MyEvent {
  public event MyEventHandler SomeEvent;

  // This is called to fire the event.
  public void OnSomeEvent() {
    if(SomeEvent != null)
      SomeEvent();
  }
}

class X {
  public void Xhandler() {
    Console.WriteLine("Event received by X object");
  }
}

class Y {
  public void Yhandler() {
    Console.WriteLine("Event received by Y object");
  }
}

class EventDemo2 {
  static void Handler() {
    Console.WriteLine("Event received by EventDemo");
  }

  static void Main() {
    MyEvent evt = new MyEvent();
    X xOb = new X();
    Y yOb = new Y();
```

```
    // Add handlers to the event list.
    evt.SomeEvent += Handler;
    evt.SomeEvent += xOb.Xhandler;
    evt.SomeEvent += yOb.Yhandler;

    // Fire the event.
    evt.OnSomeEvent();
    Console.WriteLine();

    // Remove a handler.
    evt.SomeEvent -= xOb.Xhandler;
    evt.OnSomeEvent();
  }
}
```

The output from the program is shown here:

```
Event received by EventDemo
Event received by X object
Event received by Y object

Event received by EventDemo
Event received by Y object
```

This example creates two additional classes, called **X** and **Y**, which also define event handlers compatible with **MyEventHandler**. Thus, these handlers can also become part of the event chain. Notice that the handlers in **X** and **Y** are not **static**. This means that objects of each must be created, and the handler linked to each instance must be added to the event chain. The differences between instance and **static** handlers is examined in the next section.

Instance Methods vs. Static Methods as Event Handlers

Although both instance methods and **static** methods can be used as event handlers, they do differ in one important way. When a **static** method is used as a handler, an event notification applies to the class. When an instance method is used as an event handler, events are sent to specific object instances. Thus, each object of a class that wants to receive an event notification must register individually. In practice, most event handlers are instance methods, but, of course, this is subject to the specific application. Let's look at an example of each.

The following program creates a class called **X** that defines an instance method as an event handler. This means that each **X** object must register individually to receive events. To demonstrate this fact, the program multicasts an event to three objects of type **X**.

```
/* Individual objects receive notifications when instance
   event handlers are used. */

using System;

// Declare a delegate type for an event.
delegate void MyEventHandler();

// Declare a class that contains an event.
class MyEvent {
```

```
  public event MyEventHandler SomeEvent;

  // This is called to fire the event.
  public void OnSomeEvent() {
    if(SomeEvent != null)
      SomeEvent();
  }
}

class X {
  int id;

  public X(int x) { id = x; }

  // This is an instance method that will be used as an event handler.
  public void Xhandler() {
    Console.WriteLine("Event received by object " + id);
  }
}

class EventDemo3 {
  static void Main() {
    MyEvent evt = new MyEvent();
    X o1 = new X(1);
    X o2 = new X(2);
    X o3 = new X(3);

    evt.SomeEvent += o1.Xhandler;
    evt.SomeEvent += o2.Xhandler;
    evt.SomeEvent += o3.Xhandler;

    // Fire the event.
    evt.OnSomeEvent();
  }
}
```

The output from this program is shown here:

```
Event received by object 1
Event received by object 2
Event received by object 3
```

As the output shows, each object registers its interest in an event separately, and each receives a separate notification.

Alternatively, when a **static** method is used as an event handler, events are handled independently of any object, as the following program shows:

```
/* A class receives the notification when
   a static method is used as an event handler. */

using System;

// Declare a delegate type for an event.
```

```
delegate void MyEventHandler();

// Declare a class that contains an event.
class MyEvent {
  public event MyEventHandler SomeEvent;

  // This is called to fire the event.
  public void OnSomeEvent() {
    if(SomeEvent != null)
      SomeEvent();
  }
}

class X {

  /* This is a static method that will be used as
     an event handler. */
  public static void Xhandler() {
    Console.WriteLine("Event received by class.");
  }
}

class EventDemo4 {
  static void Main() {
    MyEvent evt = new MyEvent();

    evt.SomeEvent += X.Xhandler;

    // Fire the event.
    evt.OnSomeEvent();
  }
}
```

The output from this program is shown here:

```
Event received by class.
```

In the program, notice that no object of type **X** is ever created. However, since **Xhandler( )** is a **static** method of **X**, it can be attached to **SomeEvent** and executed when **OnSomeEvent( )** is called.

Using Event Accessors

The form of **event** used in the preceding examples created events that automatically manage the event handler invocation list, including the adding and subtracting of event handlers to and from the list. Thus, you did not need to implement any of the list management functionality yourself. Because they manage the details for you, these types of events are by far the most commonly used. It is possible, however, to provide the event handler list operations yourself, perhaps to implement some type of specialized event storage mechanism.

To take control of the event handler list, you will use an expanded form of the **event** statement, which allows the use of *event accessors.* The accessors give you control over how the event handler list is implemented. This form is shown here:

```
event event-delegate event-name {
  add {
    // code to add an event to the chain
  }

  remove {
    // code to remove an event from the chain
  }
}
```

This form includes the two event accessors **add** and **remove**. The **add** accessor is called when an event handler is added to the event chain, by using **+=**. The **remove** accessor is called when an event handler is removed from the chain, by using **– =**.

When **add** or **remove** is called, it receives the handler to add or remove as a parameter. As with other types of accessors, this parameter is called **value**. By implementing **add** and **remove**, you can define a custom event-handler storage scheme. For example, you could use an array, a stack, or a queue to store the handlers.

Here is an example that uses the accessor form of **event**. It uses an array to hold the event handlers. Because the array is only three elements long, only three event handlers can be held in the chain at any one time.

```
// Create a custom means of managing the event invocation list.

using System;

// Declare a delegate type for an event.
delegate void MyEventHandler();

// Declare a class that holds up to 3 events.
class MyEvent {
  MyEventHandler[] evnt = new MyEventHandler[3];

  public event MyEventHandler SomeEvent {
    // Add an event to the list.
    add {
      int i;

      for(i=0; i < 3; i++)
        if(evnt[i] == null) {
          evnt[i] = value;
          break;
        }
      if (i == 3) Console.WriteLine("Event list full.");
    }

    // Remove an event from the list.
    remove {
      int i;

      for(i=0; i < 3; i++)
        if(evnt[i] == value) {
```

```
          evnt[i] = null;
          break;
        }
      if (i == 3) Console.WriteLine("Event handler not found.");
    }
  }

  // This is called to fire the events.
  public void OnSomeEvent() {
      for(int i=0; i < 3; i++)
        if(evnt[i] != null) evnt[i]();
  }

}

// Create some classes that use MyEventHandler.
class W {
  public void Whandler() {
    Console.WriteLine("Event received by W object");
  }
}

class X {
  public void Xhandler() {
    Console.WriteLine("Event received by X object");
  }
}

class Y {
  public void Yhandler() {
    Console.WriteLine("Event received by Y object");
  }
}

class Z {
  public void Zhandler() {
    Console.WriteLine("Event received by Z object");
  }
}

class EventDemo5 {
  static void Main() {
    MyEvent evt = new MyEvent();
    W wOb = new W();
    X xOb = new X();
    Y yOb = new Y();
    Z zOb = new Z();

    // Add handlers to the event list.
    Console.WriteLine("Adding events.");
    evt.SomeEvent += wOb.Whandler;
    evt.SomeEvent += xOb.Xhandler;
    evt.SomeEvent += yOb.Yhandler;
```

```
    // Can't store this one -- full.
    evt.SomeEvent += zOb.Zhandler;
    Console.WriteLine();

    // Fire the events.
    evt.OnSomeEvent();
    Console.WriteLine();

    // Remove a handler.
    Console.WriteLine("Remove xOb.Xhandler.");
    evt.SomeEvent -= xOb.Xhandler;
    evt.OnSomeEvent();

    Console.WriteLine();

    // Try to remove it again.
    Console.WriteLine("Try to remove xOb.Xhandler again.");
    evt.SomeEvent -= xOb.Xhandler;
    evt.OnSomeEvent();

    Console.WriteLine();

    // Now, add Zhandler.
    Console.WriteLine("Add zOb.Zhandler.");
    evt.SomeEvent += zOb.Zhandler;
    evt.OnSomeEvent();

  }
}
```

The output from the program is shown here:

```
Adding events.
Event list full.

Event received by W object
Event received by X object
Event received by Y object

Remove xOb.Xhandler.
Event received by W object
Event received by Y object

Try to remove xOb.Xhandler again.
Event handler not found.
Event received by W object
Event received by Y object

Add zOb.Zhandler.
Event received by W object
Event received by Z object
Event received by Y object
```

Let's examine this program closely. First, an event handler delegate called **MyEventHandler** is defined. Next, the class **MyEvent** is declared. It begins by defining a three-element array of event handlers called **evnt**, as shown here:

```
MyEventHandler[] evnt = new MyEventHandler[3];
```

This array will be used to store the event handlers that are added to the event chain. The elements in **evnt** are initialized to **null** by default.

Next, the event **SomeEvent** is declared. It uses the accessor form of the **event** statement, as shown here:

```
public event MyEventHandler SomeEvent {
  // Add an event to the list.
  add {
    int i;

    for(i=0; i < 3; i++)
      if(evnt[i] == null) {
        evnt[i] = value;
        break;
      }
    if (i == 3) Console.WriteLine("Event queue full.");
  }

  // Remove an event from the list.
  remove {
    int i;

    for(i=0; i < 3; i++)
      if(evnt[i] == value) {
        evnt[i] = null;
        break;
      }
    if (i == 3) Console.WriteLine("Event handler not found.");
  }
}
```

When an event handler is added, **add** is called and a reference to the handler (contained in **value**) is put into the first unused (that is, null) element of **evnt**. If no element is free, then an error is reported. (Of course, throwing an exception when the list is full would be a better approach for real-world code.) Since **evnt** is only three elements long, only three event handlers can be stored. When an event handler is removed, **remove** is called and the **evnt** array is searched for the reference to the handler passed in **value**. If it is found, its element in the array is assigned **null**, thus removing the handler from the list.

When an event is fired, **OnSomeEvent( )** is called. It cycles through the **evnt** array, calling each event handler in turn.

As the preceding example shows, it is relatively easy to implement a custom event-handler storage mechanism if one is needed. For most applications, though, the default storage provided by the non-accessor form of **event** is better. The accessor-based form of **event** can be useful in certain specialized situations, however. For example, if you have a program in which event handlers need to be executed in order of their priority and not in the order in which they are added to the chain, then you could use a priority queue to store the handlers.

NOTE *In multithreaded applications, you will usually need to synchronize access to the event accessors. For information on multithreaded programming, see Chapter 23.*

Miscellaneous Event Features

Events can be specified in interfaces. Implementing classes must supply the event.

Events can also be specified as **abstract**. A derived class must implement the event. Accessor-based events cannot, however, be **abstract**.

An event can be specified as **sealed**.

Finally, an event can be virtual, which means that it can be overridden in a derived class.

Use Anonymous Methods and Lambda Expressions with Events

Anonymous methods and lambda expressions are especially useful when working with events because often the event handler is not called by any code other than the event handling mechanism. As a result, there is usually no reason to create a standalone method. Thus, the use of lambda expressions or anonymous methods can significantly streamline event handling code.

Since lambda expressions are now the preferred approach, we will start there. Here is an example that uses a lambda expression as an event handler:

```
// Use a lambda expression as an event handler.
using System;

// Declare a delegate type for an event.
delegate void MyEventHandler(int n);

// Declare a class that contains an event.
class MyEvent {
  public event MyEventHandler SomeEvent;

  // This is called to fire the event.
  public void OnSomeEvent(int n) {
    if(SomeEvent != null)
      SomeEvent(n);
  }
}

class LambdaEventDemo {
  static void Main() {
    MyEvent evt = new MyEvent();

    // Use a lambda expression as an event handler.
    evt.SomeEvent += (n) =>
      Console.WriteLine("Event received. Value is " + n);

    // Fire the event twice.
    evt.OnSomeEvent(1);
    evt.OnSomeEvent(2);
  }
}
```

The output is shown here:

```
Event received. Value is 1
Event received. Value is 2
```

In the program, pay special attention to the way the lambda expression is used as an event handler, as shown here:

```
evt.SomeEvent += (n) =>
  Console.WriteLine("Event received. Value is " + n);
```

The syntax for using a lambda expression event handler is the same as that for using a lambda expression with any other type of delegate.

Although lambda expressions are now the preferred way to construct an anonymous function, you can still use an anonymous method as an event handler if you so choose. For example, here is the event handler from the previous example rewritten to use an anonymous method:

```
// Use an anonymous method as an event handler.
evt.SomeEvent += delegate(int n)  {
  Console.WriteLine("Event received. Value is" + n);
};
```

As you can see, the syntax for using an anonymous event handler is the same as that for any anonymous method.

.NET Event Guidelines

C# allows you to write any type of event you desire. However, for component compatibility with the .NET Framework, you will need to follow the guidelines Microsoft has established for this purpose. At the core of these guidelines is the requirement that event handlers have two parameters. The first is a reference to the object that generated the event. The second is a parameter of type **EventArgs** that contains any other information required by the handler. Thus, .NET-compatible event handlers will have this general form:

void *handler*(object *source*, EventArgs *arg*) {
  // ...
}

Typically, the *source* parameter is passed **this** by the calling code. The **EventArgs** parameter contains additional information and can be ignored if it is not needed.

The **EventArgs** class itself does not contain fields that you use to pass additional data to a handler. Instead, **EventArgs** is used as a base class from which you will derive a class that contains the necessary fields. **EventArgs** does include one **static** field called **Empty**, which is an **EventArgs** object that contains no data.

Here is an example that creates a .NET-compatible event:

```
// A .NET-compatible event.

using System;

// Derive a class from EventArgs.
class MyEventArgs : EventArgs {
  public int EventNum;
```

```
}

// Declare a delegate type for an event.
delegate void MyEventHandler(object source, MyEventArgs arg);

// Declare a class that contains an event.
class MyEvent {
  static int count = 0;

  public event MyEventHandler SomeEvent;

  // This fires SomeEvent.
  public void OnSomeEvent() {
    MyEventArgs arg = new MyEventArgs();

    if(SomeEvent != null) {
      arg.EventNum = count++;
      SomeEvent(this, arg);
    }
  }
}

class X {
  public void Handler(object source, MyEventArgs arg) {
    Console.WriteLine("Event " + arg.EventNum +
                      " received by an X object.");
    Console.WriteLine("Source is " + source);
    Console.WriteLine();
  }
}

class Y {
  public void Handler(object source, MyEventArgs arg) {
    Console.WriteLine("Event " + arg.EventNum +
                      " received by a Y object.");
    Console.WriteLine("Source is " + source);
    Console.WriteLine();
  }
}

class EventDemo6 {
  static void Main() {
    X ob1 = new X();
    Y ob2 = new Y();
    MyEvent evt = new MyEvent();

    // Add Handler() to the event list.
    evt.SomeEvent += ob1.Handler;
    evt.SomeEvent += ob2.Handler;

    // Fire the event.
    evt.OnSomeEvent();
    evt.OnSomeEvent();
  }
}
```

Here is the output:

```
Event 0 received by an X object.
Source is MyEvent

Event 0 received by a Y object.
Source is MyEvent

Event 1 received by an X object.
Source is MyEvent

Event 1 received by a Y object.
Source is MyEvent
```

In this example, **MyEventArgs** is derived from **EventArgs**. **MyEventArgs** adds just one field of its own: **EventNum**. The event handler delegate **MyEventHandler** now takes the two parameters required by the .NET Framework. As explained, the first is an object reference to the generator of the event. The second is a reference to **EventArgs** or a class derived from **EventArgs**. The event handlers in the **X** and **Y** classes, **Handler( )**, also have the same types of parameters.

Inside **MyEvent**, a **MyEventHandler** called **SomeEvent** is declared. In the **OnSomeEvent( )** method, **SomeEvent** is called with the first argument being **this**, and the second argument being a **MyEventArgs** instance. Thus, the proper arguments are passed to **MyEventHandler** to fulfill the requirements for .NET compatibility.

Use EventHandler

For many events, the **EventArgs** parameter is unused. To help facilitate the creation of code in these situations, the .NET Framework includes a built-in delegate type called **EventHandler**, which can be used to declare event handlers in which no extra information is needed. Here is an example that uses **EventHandler**:

```
// Use the built-in EventHandler delegate.

using System;

// Declare a class that contains an event.
class MyEvent {
  public event EventHandler SomeEvent; // uses EventHandler delegate

  // This is called to fire SomeEvent.
  public void OnSomeEvent() {
    if(SomeEvent != null)
      SomeEvent(this, EventArgs.Empty);
  }
}

class EventDemo7 {
  static void Handler(object source, EventArgs arg) {
    Console.WriteLine("Event occurred");
    Console.WriteLine("Source is " + source);
  }
```

```
  static void Main() {
    MyEvent evt = new MyEvent();

    // Add Handler() to the event list.
    evt.SomeEvent += Handler;

    // Fire the event.
    evt.OnSomeEvent();
  }
}
```

In this case, the **EventArgs** parameter is unused and is passed the placeholder object **EventArgs.Empty**. The output is shown here:

```
Event occurred
Source is MyEvent
```

Applying Events: A Case Study

Events are frequently used in message-based environments such as Windows. In such an environment, a program simply waits until it receives a message, and then it takes the appropriate action. Such an architecture is well suited for C#-style event handling because it is possible to create event handlers for various messages and then simply invoke a handler when a message is received. For example, the left-button mouse click message could be tied to an event called **LButtonClick**. When a left-button click is received, a method called **OnLButtonClick( )** can be called, and all registered handlers will be notified.

Although developing a Windows program that demonstrates this approach is beyond the scope of this chapter, it is possible to give an idea of how such an approach would work. The following program creates an event handler that processes keystrokes. The event is called **KeyPress**, and each time a key is pressed, the event is fired by calling **OnKeyPress( )**. Notice that .NET-compatible events are created and that lambda expressions provide the event handlers.

```
// A keypress event example.

using System;

// Derive a custom EventArgs class that holds the key.
class KeyEventArgs : EventArgs {
  public char ch;
}

// Declare a delegate type for an event.
delegate void KeyHandler(object source, KeyEventArgs arg);

// Declare a keypress event class.
class KeyEvent {
  public event KeyHandler KeyPress;

  // This is called when a key is pressed.
  public void OnKeyPress(char key) {
```

```
    KeyEventArgs k = new KeyEventArgs();

    if(KeyPress != null) {
      k.ch = key;
      KeyPress(this, k);
    }
  }
}

// Demonstrate KeyEvent.
class KeyEventDemo {
  static void Main() {
    KeyEvent kevt = new KeyEvent();
    ConsoleKeyInfo key;
    int count = 0;

    // Use a lambda expression to display the keypress.
    kevt.KeyPress += (source, arg) =>
      Console.WriteLine(" Received keystroke: " + arg.ch);

    // Use a lambda expression to count keypresses.
    kevt.KeyPress += (source, arg) =>
      count++; // count is an outer variable

    Console.WriteLine("Enter some characters. " +
                      "Enter a period to stop.");
    do {
      key = Console.ReadKey();
      kevt.OnKeyPress(key.KeyChar);
    } while(key.KeyChar != '.');

    Console.WriteLine(count + " keys pressed.");
  }
}
```

Here is a sample run:

```
Enter some characters. Enter a period to stop.
t Received keystroke: t
e Received keystroke: e
s Received keystroke: s
t Received keystroke: t
. Received keystroke: .
5 keys pressed.
```

The program begins by deriving a class from **EventArgs** called **KeyEventArgs**, which is used to pass a keystroke to an event handler. Next, a delegate called **KeyHandler** defines the event handler for keystroke events. The class **KeyEvent** encapsulates the keypress event. It defines the event **KeyPress**.

In **Main( )**, a **KeyEvent** object called **kevt** is created. Next, an event handler based on a lambda expression is added to **kvet.KeyPress** that displays each key as it is entered, as shown here:

```
kevt.KeyPress += (source, arg) =>
  Console.WriteLine(" Received keystroke: " + arg.ch);
```

Next, another lambda expression–based handler is added to **kvet.KeyPress** by the following code. It counts the number of keypresses.

```
kevt.KeyPress += (source, arg) =>
  count++; // count is an outer variable
```

Notice that **count** is a local variable declared in **Main( )** that is initialized to zero.

Next, a loop is started that calls **kevt.OnKeyPress( )** when a key is pressed. This causes the registered event handlers to be notified. When the loop ends, the number of keypresses is displayed. Although quite simple, this example illustrates the essence of event handling. The same basic approach will be used for other event handling situations. Of course, in some cases, anonymous event handlers will not be appropriate and named methods will need to be employed.

This page intentionally left blank

16
CHAPTER

Namespaces, the Preprocessor, and Assemblies

This chapter discusses three C# features that give you greater control over the organization and accessibility of a program. These are namespaces, the preprocessor, and assemblies.

Namespaces

The namespace was mentioned briefly in Chapter 2 because it is a concept fundamental to C#. In fact, every C# program makes use of a namespace in one way or another. We didn't need to examine namespaces in detail before now because C# automatically provides a default, global namespace for your program. Thus, the programs in earlier chapters simply used the global namespace. In the real world, however, many programs will need to create their own namespaces or interact with other namespaces. Here, they are examined in detail.

A *namespace* defines a declarative region that provides a way to keep one set of names separate from another. In essence, names declared in one namespace will not conflict with the same names declared in another. The namespace used by the .NET Framework library (which is the C# library) is **System**. This is why you have included

```
using System;
```

near the top of every program. As explained in Chapter 14, the I/O classes are defined within a namespace subordinate to **System** called **System.IO**. There are many other namespaces subordinate to **System** that hold other parts of the C# library.

Namespaces are important because there has been an explosion of variable, method, property, and class names over the past few years. These include library routines, third-party code, and your own code. Without namespaces, all of these names would compete for slots in the global namespace and conflicts would arise. For example, if your program defined a class called **Finder**, it could conflict with another class called **Finder** supplied by a third-party library that your program uses. Fortunately, namespaces prevent this type of problem because a namespace restricts the visibility of names declared within it.

Declaring a Namespace

A namespace is declared using the **namespace** keyword. The general form of **namespace** is shown here:

```
namespace name {
  // members
}
```

Here, *name* is the name of the namespace. A namespace declaration defines a scope. Anything declared immediately inside the namespace is in scope throughout the namespace. Within a namespace, you can declare classes, structures, delegates, enumerations, interfaces, or another namespace.

Here is an example of a **namespace** that creates a namespace called **Counter**. It localizes the name used to implement a simple countdown counter class called **CountDown**.

```
// Declare a namespace for counters.

namespace Counter {
  // A simple countdown counter.
  class CountDown {
    int val;

    public CountDown(int n) {
      val = n;
   }

    public void Reset(int n) {
      val = n;
    }

    public int Count() {
      if(val > 0) return val--;
      else return 0;
    }
  }
} // This is the end of the Counter namespace.
```

Notice how the class **CountDown** is declared within the scope defined by the **Counter** namespace. To follow along with the example, put this code into a file called **Counter.cs**.

Here is a program that demonstrates the use of the **Counter** namespace:

```
// Demonstrate the Counter namespace.

using System;

class NSDemo {
  static void Main() {
    // Notice how CountDown is qualified by Counter.
    Counter.CountDown cd1 = new Counter.CountDown(10);
    int i;

    do {
      i = cd1.Count();
      Console.Write(i + " ");
```

```
    } while(i > 0);
    Console.WriteLine();

    // Again, notice how CountDown is qualified by Counter.
    Counter.CountDown cd2 = new Counter.CountDown(20);

    do {
      i = cd2.Count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    cd2.Reset(4);
    do {
      i = cd2.Count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();
  }
}
```

The output from the program is shown here:

```
10 9 8 7 6 5 4 3 2 1 0
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
4 3 2 1 0
```

To compile this program, you must include both the preceding code and the code contained in the **Counter** namespace. Assuming you called the preceding code **NSDemo.cs** and put the source code for the **Counter** namespace into a file called **Counter.cs** as mentioned earlier, then you can use this command line to compile the program:

```
csc NSDemo.cs counter.cs
```

Some important aspects of this program warrant close examination. First, since **CountDown** is declared within the **Counter** namespace, when an object is created, **CountDown** must be qualified with **Counter**, as shown here:

```
Counter.CountDown cd1 = new Counter.CountDown(10);
```

This rule can be generalized. Whenever you use a member of a namespace, you must qualify it with the namespace name. If you don't, the member of the namespace won't be found by the compiler.

Second, once an object of type **Counter** has been created, it is not necessary to further qualify it or any of its members with the namespace. Thus, **cd1.Count( )** can be called directly without namespace qualification, as this line shows:

```
i = cd1.Count();
```

Third, for the sake of illustration, this example uses two separate files. One holds the **Counter** namespace and the other holds the **NSDemo** program. However, both could have been contained in the same file. Furthermore, a single file can contain two or more named namespaces, with each namespace defining its own declarative region. When a named

namespace ends, the outer namespace resumes, which in the case of the **Counter** is the global namespace. For clarity, subsequent examples will show all namespaces required by a program within the same file, but remember that separate files would be equally valid (and more commonly used in production code).

REMEMBER *For clarity, the remaining namespace examples in this chapter show all namespaces required by a program within the same file. In real-world code, however, a namespace will often be defined in its own file, as the preceding example illustrates.*

Namespaces Prevent Name Conflicts

The key point about a namespace is that names declared within it won't conflict with similar names declared outside of it. For example, the following program defines two namespaces. The first is **Counter**, shown earlier. The second is called **Counter2**. Both contain classes called **CountDown**, but because they are in separate namespaces, the two classes do not conflict. Also notice how both namespaces are specified within the same file. As just explained, a single file can contain multiple namespace declarations. Of course, separate files for each namespace could also have been used.

```
// Namespaces prevent name conflicts.

using System;

// Declare the Counter namespace.
namespace Counter {
  // A simple countdown counter.
  class CountDown {
    int val;

    public CountDown(int n) {
      val = n;
    }

    public void Reset(int n) {
      val = n;
    }

    public int Count() {
      if(val > 0) return val--;
      else return 0;
    }
  }
}

// Declare the Counter2 namespace.
namespace Counter2 {
  /* This CountDown is in the Counter2 namespace and
     does not conflict with the one in Counter. */
  class CountDown {
    public void Count() {
      Console.WriteLine("This is Count() in the " +
                        "Counter2 namespace.");
    }
```

```
  }
}

class NSDemo2 {
  static void Main() {
    // This is CountDown in the Counter namespace.
    Counter.CountDown cd1 = new Counter.CountDown(10);

    // This is CountDown in the Counter2 namespace.
    Counter2.CountDown cd2 = new Counter2.CountDown();

    int i;

    do {
      i = cd1.Count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    cd2.Count();
  }
}
```

The output is shown here:

```
10 9 8 7 6 5 4 3 2 1 0
This is Count() in the Counter2 namespace.
```

As the output confirms, the **CountDown** class inside **Counter** is separate from the **CountDown** class in the **Counter2** namespace, and no name conflicts arise. Although this example is quite simple, it is easy to see how putting classes into a namespace helps prevent name conflicts between your code and code written by others.

using

If your program includes frequent references to the members of a namespace, having to specify the namespace each time you need to refer to one quickly becomes tedious. The **using** directive alleviates this problem. Throughout this book, you have been using it to bring the C# **System** namespace into view, so you are already familiar with it. As you would expect, **using** can also be employed to bring namespaces that you create into view.

There are two forms of the **using** directive. The first is shown here:

using *name*;

Here, *name* specifies the name of the namespace you want to access. This is the form of **using** that you have already seen. All of the members defined within the specified namespace are brought into view and can be used without qualification. A **using** directive must be specified at the top of each file, prior to any other declarations, or at the start of a namespace body.

The following program reworks the counter example to show how you can employ **using** to bring a namespace that you create into view:

```
// Demonstrate the using directive.

using System;
```

```
// Bring Counter into view.
using Counter;

// Declare a namespace for counters.
namespace Counter {
  // A simple countdown counter.
  class CountDown {
    int val;

    public CountDown(int n) {
      val = n;
    }

    public void Reset(int n) {
      val = n;
    }

    public int Count() {
      if(val > 0) return val--;
      else return 0;
    }
  }
}

class NSDemo3 {
  static void Main() {
    // now, CountDown can be used directly.
    CountDown cd1 = new CountDown(10);
    int i;

    do {
      i = cd1.Count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    CountDown cd2 = new CountDown(20);

    do {
      i = cd2.Count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    cd2.Reset(4);
    do {
      i = cd2.Count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();
  }
}
```

This version of the program contains two important changes. The first is this **using** statement, near the top of the program:

```
using Counter;
```

This brings the **Counter** namespace into view. The second change is that it is no longer necessary to qualify **CountDown** with **Counter**, as this statement in **Main( )** shows:

```
CountDown cd1 = new CountDown(10);
```

Because **Counter** is now in view, **CountDown** can be used directly.

The program illustrates one other important point: Using one namespace does not override another. When you bring a namespace into view, it simply lets you use its contents without qualification. Thus, in the example, both **System** and **Counter** have been brought into view.

A Second Form of using

The **using** directive has a second form that creates another name, called an alias, for a type or a namespace. This form is shown here:

using *alias* = *name*;

Here, *alias* becomes another name for the type (such as a class type) or namespace specified by *name*. Once the alias has been created, it can be used in place of the original name.

Here the example from the preceding section has been reworked so that an alias for **Counter.CountDown** called **MyCounter** is created:

```
// Demonstrate a using alias.

using System;

// Create an alias for Counter.CountDown.
using MyCounter = Counter.CountDown;

// Declare a namespace for counters.
namespace Counter {
  // A simple countdown counter.
  class CountDown {
    int val;

    public CountDown(int n) {
      val = n;
    }

    public void Reset(int n) {
      val = n;
    }

    public int Count() {
      if(val > 0) return val--;
      else return 0;
    }
  }
}

class NSDemo4 {
```

```
  static void Main() {
    // Here, MyCounter is used as a name for Counter.CountDown.
    MyCounter cd1 = new MyCounter(10);
    int i;

    do {
      i = cd1.Count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    MyCounter cd2 = new MyCounter(20);

    do {
      i = cd2.Count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    cd2.Reset(4);
    do {
      i = cd2.Count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();
  }
}
```

The **MyCounter** alias is created using this statement:

```
using MyCounter = Counter.CountDown;
```

Once **MyCounter** has been specified as another name for **Counter.CountDown**, it can be used to declare objects without any further namespace qualification. For example, in the program, this line

```
    MyCounter cd1 = new MyCounter(10);
```

creates a **CountDown** object.

Namespaces Are Additive

There can be more than one namespace declaration of the same name. This allows a namespace to be split over several files or even separated within the same file. For example, the following program defines two **Counter** namespaces. One contains the **CountDown** class. The other contains the **CountUp** class. When compiled, the contents of both **Counter** namespaces are added together.

```
// Namespaces are additive.

using System;

// Bring Counter into view.
using Counter;
```

```
// Here is one Counter namespace.
namespace Counter {
  // A simple countdown counter.
  class CountDown {
    int val;

    public CountDown(int n) {
      val = n;
    }

    public void Reset(int n) {
      val = n;
    }

    public int Count() {
      if(val > 0) return val--;
      else return 0;
    }
  }
}

// Here is another Counter namespace.
namespace Counter {
  // A simple count-up counter.
  class CountUp {
    int val;
    int target;

    public int Target {
      get{
        return target;
      }
    }

    public CountUp(int n) {
      target = n;
      val = 0;
    }

    public void Reset(int n) {
      target = n;
      val = 0;
    }

    public int Count() {
      if(val < target) return val++;
      else return target;
    }
  }
}

class NSDemo5 {
  static void Main() {
    CountDown cd = new CountDown(10);
```

```
    CountUp cu = new CountUp(8);
    int i;

    do {
      i = cd.Count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    do {
      i = cu.Count();
      Console.Write(i + " ");
    } while(i < cu.Target);

  }
}
```

This program produces the following output:

```
10 9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8
```

Notice one other thing: The directive

```
using Counter;
```

brings into view the entire contents of the **Counter** namespace. Thus, both **CountDown** and **CountUp** can be referred to directly, without namespace qualification. It doesn't matter that the **Counter** namespace was split into two parts.

Namespaces Can Be Nested

One namespace can be nested within another. Consider this program:

```
// Namespaces can be nested.

using System;

namespace NS1 {
  class ClassA {
     public ClassA() {
       Console.WriteLine("constructing ClassA");
    }
  }
  namespace NS2 { // a nested namespace
    class ClassB {
       public ClassB() {
         Console.WriteLine("constructing ClassB");
       }
    }
  }
}

class NestedNSDemo {
  static void Main() {
```

```
    NS1.ClassA a = new NS1.ClassA();

 // NS2.ClassB b = new NS2.ClassB(); // Error!!! NS2 is not in view

    NS1.NS2.ClassB b = new NS1.NS2.ClassB(); // this is right
  }
}
```

This program produces the following output:

```
constructing ClassA
constructing ClassB
```

In the program, the namespace **NS2** is nested within **NS1**. Thus, to refer to **ClassB**, you must qualify it with both the **NS1** and **NS2** namespaces. **NS2**, by itself, is insufficient. As shown, the namespace names are separated by a period. Therefore, to refer to **ClassB** within **Main( )**, you must use **NS1.NS2.ClassB**.

Namespaces can be nested by more than two levels. When this is the case, a member in a nested namespace must be qualified with all of the enclosing namespace names.

You can specify a nested namespace using a single **namespace** statement by separating each namespace with a period. For example,

```
namespace OuterNS {
  namespace InnerNS {
    // ...
  }
}
```

can also be specified like this:

```
namespace OuterNS.InnerNS {
  // ...
}
```

The Global Namespace

If you don't declare a namespace for your program, then the default global namespace is used. This is why you have not needed to use **namespace** for the programs in the preceding chapters. Although the global namespace is convenient for the short, sample programs found in this book, most real-world code will be contained within a declared namespace. The main reason for encapsulating your code within a declared namespace is that it prevents name conflicts. Namespaces are another tool that you have to help you organize programs and make them viable in today's complex, networked environment.

Using the :: Namespace Alias Qualifier

Although namespaces help prevent name conflicts, they do not completely eliminate them. One way that a conflict can still occur is when the same name is declared within two different namespaces, and you then try to bring both namespaces into view. For example, assume that two different namespaces contain a class called **MyClass**. If you attempt to bring these two namespaces into view via **using** statements, **MyClass** in the first namespace will conflict with **MyClass** in the second namespace, causing an ambiguity error. In this situation, you can use the **::** *namespace alias qualifier* to explicitly specify which namespace is intended.

The :: operator has this general form.

namespace-alias::*identifier*

Here, *namespace-alias* is the name of a namespace alias and *identifier* is the name of a member of that namespace.

To understand why the namespace alias qualifier is needed, consider the following program. It creates two namespaces, **Counter** and **AnotherCounter**, and both declare a class called **CountDown**. Furthermore, both namespaces are brought into view by **using** statements. Finally, in **Main( )**, an attempt is made to instantiate an object of type **CountDown**.

```
// Demonstrate why the :: qualifier is needed.
//
// This program will not compile.

using System;

// Use both the Counter and AnotherCounter namespace.
using Counter;
using AnotherCounter;

// Declare a namespace for counters.
namespace Counter {
  // A simple countdown counter.
  class CountDown {
    int val;

    public CountDown(int n) {
      val = n;
    }

    // ...
  }
}

// Declare another namespace for counters.
namespace AnotherCounter {
  // Declare another class called CountDown, which
  // is in the AnotherCounter namespace.
  class CountDown {
    int val;

    public CountDown(int n) {
      val = n;
    }

    // ...
  }
}

class WhyAliasQualifier {
  static void Main() {
    int i;
```

```
    // The following line is inherently ambiguous!
    // Does it refer to CountDown in Counter or
    // to CountDown in AnotherCounter?
    CountDown cd1 = new CountDown(10); // Error! ! !

    // ...
  }
}
```

If you try to compile this program, you will receive an error message stating that this line in **Main( )** is ambiguous:

```
CountDown cd1 = new CountDown(10); // Error! ! !
```

The trouble is that both namespaces, **Counter** and **AnotherCounter**, declare a class called **CountDown**, and both namespaces have been brought into view. Thus, to which version of **CountDown** does the preceding declaration refer? The **::** qualifier was designed to handle these types of problems.

To use the **::**, you must first define an alias for the namespace you want to qualify. Then, simply qualify the ambiguous element with the alias. For example, here is one way to fix the preceding program:

```
// Demonstrate the :: qualifier.

using System;

using Counter;
using AnotherCounter;

// Give Counter an alias called Ctr.
using Ctr = Counter;

// Declare a namespace for counters.
namespace Counter {
  // A simple countdown counter.
  class CountDown {
    int val;

    public CountDown(int n) {
      val = n;
    }

    // ...
  }
}

// Another counter namespace.
namespace AnotherCounter {
  // Declare another class called CountDown, which
  // is in the AnotherCounter namespace.
  class CountDown {
    int val;
```

```
    public CountDown(int n) {
      val = n;
    }

    // ...
  }
}

class AliasQualifierDemo {
  static void Main() {

    // Here, the :: operator to resolve
    // tells the compiler to use the CountDown
    // that is in the Counter namespace.
    Ctr::CountDown cd1 = new Ctr::CountDown(10);

    // ...
  }
}
```

In this version, the alias **Ctr** is specified for **Counter** by the following line:

```
using Ctr = Counter;
```

Then, inside **Main( )**, this alias is used to qualify **CountDown**, as shown here:

```
Ctr::CountDown cd1 = new Ctr::CountDown(10);
```

The use of the **::** qualifier removes the ambiguity because it specifies that the **CountDown** in **Ctr** (which stands for **Counter**) is desired, and the program now compiles.

You can use the **::** qualifier to refer to the global namespace by using the predefined identifier **global**. For example, in the following program, a class called **CountDown** is declared in both the **Counter** namespace and in the global namespace. To access the version of **CountDown** in the global namespace, the predefined alias **global** is used.

```
// Use the global alias.

using System;

// Give Counter an alias called Ctr.
using Ctr = Counter;

// Declare a namespace for counters.
namespace Counter {
  // A simple countdown counter.
  class CountDown {
    int val;

    public CountDown(int n) {
      val = n;
    }

    // ...
  }
```

```
}

// Declare another class called CountDown, which
// is in the global namespace.
class CountDown {
  int val;

  public CountDown(int n) {
    val = n;
  }

  // ...
}

class GlobalAliasQualifierDemo {
  static void Main() {

    // Here, the :: qualifier tells the compiler
    // to use the CountDown in the Counter namespace.
    Ctr::CountDown cd1 = new Ctr::CountDown(10);

    // Next, create CountDown object from global namespace.
    global::CountDown cd2 = new global::CountDown(10);

    // ...
  }
}
```

Notice how the **global** identifier is used to access the version of **CountDown** in the default namespace:

```
global::CountDown cd2 = new global::CountDown(10);
```

This same general approach can be generalized to any situation in which you need to specify the default namespace.

One final point: You can also use the namespace alias qualifier with **extern** aliases, which are described in Chapter 20.

The Preprocessor

C# defines several *preprocessor directives,* which affect the way that your program's source file is interpreted by the compiler. These directives affect the text of the source file in which they occur, prior to the translation of the program into object code. The term *preprocessor directive* comes from the fact that these instructions were traditionally handled by a separate compilation phase called the *preprocessor.* Today's modern compiler technology no longer requires a separate preprocessing stage to handle the directives, but the name has stuck.

C# defines the following preprocessor directives:

#define	#elif	#else	#endif
#endregion	#error	#if	#line
#pragma	#region	#undef	#warning

All preprocessor directives begin with a **#** sign. In addition, each preprocessor directive must be on its own line.

Given C#'s modern, object-oriented architecture, there is not as much need for the preprocessor directives as there is in older languages. Nevertheless, they can be of value from time to time, especially for conditional compilation. Each directive is examined in turn.

#define

The **#define** directive defines a character sequence called a *symbol.* The existence or nonexistence of a symbol can be determined by **#if** or **#elif** and is used to control compilation. Here is the general form for #**define**:

#define *symbol*

Notice that there is no semicolon in this statement. There may be any number of spaces between the **#define** and the symbol, but once the symbol begins, it is terminated only by a newline. For example, to define the symbol **EXPERIMENTAL**, use this directive:

```
#define EXPERIMENTAL
```

NOTE *In C/C++ you can use **#define** to perform textual substitutions, such as defining a name for a value, and to create function-like macros. C# does not support these uses of **#define**. In C#, **#define** is used only to define a symbol.*

#if and #endif

The **#if** and **#endif** directives enable conditional compilation of a sequence of code based upon whether an expression involving one or more symbols evaluates to true. A symbol is true if it has been defined. It is false otherwise. Thus, if a symbol has been defined by a **#define** directive, it will evaluate as true.

The general form of **#if** is

#if *symbol-expression*
  statement sequence
#endif

If the expression following **#if** is true, the code that is between it and **#endif** is compiled. Otherwise, the intervening code is skipped. The **#endif** directive marks the end of an **#if** block.

A symbol expression can be as simple as just the name of a symbol. You can also use these operators in a symbol expression: **!**, **==**, **!=**, **&&**, and **||**. Parentheses are also allowed.

Here's an example:

```
// Demonstrate #if, #endif, and #define.

#define EXPERIMENTAL

using System;

class Test {
  static void Main() {

    #if EXPERIMENTAL
      Console.WriteLine("Compiled for experimental version.");
```

```
    #endif

    Console.WriteLine("This is in all versions.");
  }
}
```

This program displays the following:

```
Compiled for experimental version.
This is in all versions.
```

The program defines the symbol **EXPERIMENTAL**. Thus, when the **#if** is encountered, the symbol expression evaluates to true, and the first **WriteLine( )** statement is compiled. If you remove the definition of **EXPERIMENTAL** and recompile the program, the first **WriteLine( )** statement will not be compiled, because the **#if** will evaluate to false. In all cases, the second **WriteLine( )** statement is compiled because it is not part of the **#if** block.

As explained, you can use a symbol expression in an **#if**. For example,

```
// Use a symbol expression.

#define EXPERIMENTAL
#define TRIAL

using System;

class Test {
  static void Main() {

    #if EXPERIMENTAL
      Console.WriteLine("Compiled for experimental version.");
    #endif

    #if EXPERIMENTAL && TRIAL
       Console.Error.WriteLine("Testing experimental trial version.");
    #endif

    Console.WriteLine("This is in all versions.");
  }
}
```

The output from this program is shown here:

```
Compiled for experimental version.
Testing experimental trial version.
This is in all versions.
```

In this example, two symbols are defined, **EXPERIMENTAL** and **TRIAL**. The second **WriteLine( )** statement is compiled only if both are defined.

You can use the ! to compile code when a symbol is not defined. For example,

```
#if !EXPERIMENTAL
  Console.WriteLine("Code is not experimental!");
#endif
```

The call to **WriteLine( )** will be compiled only if **EXPERIMENTAL** *has not* been defined.

#else and #elif

The **#else** directive works much like the **else** that is part of the C# language: It establishes an alternative if **#if** fails. The previous example can be expanded as shown here:

```
// Demonstrate #else.

#define EXPERIMENTAL

using System;

class Test {
  static void Main() {

    #if EXPERIMENTAL
      Console.WriteLine("Compiled for experimental version.");
    #else
      Console.WriteLine("Compiled for release.");
    #endif

    #if EXPERIMENTAL && TRIAL
       Console.Error.WriteLine("Testing experimental trial version.");
    #else
       Console.Error.WriteLine("Not experimental trial version.");
    #endif

    Console.WriteLine("This is in all versions.");
  }
}
```

The output is shown here:

```
Compiled for experimental version.
Not experimental trial version.
This is in all versions.
```

Since **TRIAL** is not defined, the **#else** portion of the second conditional code sequence is used.

Notice that **#else** marks both the end of the **#if** block and the beginning of the **#else** block. This is necessary because there can only be one **#endif** associated with any **#if**. Furthermore, there can be only one **#else** associated with any **#if**.

The **#elif** directive means "else if" and establishes an if-else-if chain for multiple compilation options. **#elif** is followed by a symbol expression. If the expression is true, that block of code is compiled and no other **#elif** expressions are tested. Otherwise, the next block in the series is checked. If no **#elif** succeeds, then if there is a **#else**, the code sequence associated with the **#else** is compiled. Otherwise, no code in the entire **#if** is compiled.

The general form for **#elif** is

#if *symbol-expression*
  statement sequence
#elif *symbol-expression*
  statement sequence
#elif *symbol-expression*

statement sequence
//...
#endif

Here's an example:

```
// Demonstrate #elif.

#define RELEASE

using System;

class Test {
  static void Main() {

    #if EXPERIMENTAL
      Console.WriteLine("Compiled for experimental version.");
    #elif RELEASE
      Console.WriteLine("Compiled for release.");
    #else
      Console.WriteLine("Compiled for internal testing.");
    #endif

    #if TRIAL && !RELEASE
       Console.WriteLine("Trial version.");
    #endif

    Console.WriteLine("This is in all versions.");
  }
}
```

The output is shown here:

```
Compiled for release.
This is in all versions.
```

#undef

The **#undef** directive removes a previously defined symbol. That is, it "undefines" a symbol. The general form for **#undef** is

#undef *symbol*

Here's an example:

```
#define SMALL

#if SMALL
  // ...
#undef SMALL
// at this point SMALL is undefined.
```

After the **#undef** directive, **SMALL** is no longer defined.

#undef is used principally to allow symbols to be localized to only those sections of code that need them.

#error

The **#error** directive forces the compiler to stop compilation. It is used for debugging. The general form of the **#error** directive is

#error *error-message*

When the **#error** directive is encountered, the error message is displayed. For example, when the compiler encounters this line:

```
#error This is a test error!
```

compilation stops and the error message "This is a test error!" is displayed.

#warning

The **#warning** directive is similar to **#error**, except that a warning rather than an error is produced. Thus, compilation is not stopped. The general form of the **#warning** directive is

#warning *warning-message*

#line

The **#line** directive sets the line number and filename for the file that contains the **#line** directive. The number and the name are used when errors or warnings are output during compilation. The general form for **#line** is

#line *number* "*filename*"

where *number* is any positive integer, which becomes the new line number, and the optional *filename* is any valid file identifier, which becomes the new filename. **#line** is primarily used for debugging and special applications.

#line allows two options. The first is **default**, which returns the line numbering to its original condition. It is used like this:

```
#line default
```

The second is **hidden**. When stepping through a program, the **hidden** option allows a debugger to bypass lines between a

```
#line hidden
```

directive and the next **#line** directive that does not include the **hidden** option.

#region and #endregion

The **#region** and **#endregion** directives let you define a region that will be expanded or collapsed when using outlining in the Visual Studio IDE. The general form is shown here:

#region *text*
// code sequence
#endregion *text*

Here, *text* is an optional string.

#pragma

The **#pragma** directive gives instructions, such as specifying an option, to the compiler. It has this general form:

#pragma *option*

Here, *option* is the instruction passed to the compiler.

In C# 3.0, there are two options supported by **#pragma**. The first is **warning**, which is used to enable or disable specific compiler warnings. It has these two forms:

#pragma warning disable *warnings*

#pragma warning restore *warnings*

Here, *warnings* is a comma-separated list of warning numbers. To disable a warning, use the **disable** option. To enable a warning, use the **restore** option.

For example, this **#pragma** statement disables warning 168, which indicates when a variable is declared but not used:

```
#pragma warning disable 168
```

The second **#pragma** option is **checksum**. It is used to generate checksums for ASP.NET projects. It has this general form.

#pragma checksum "*filename*" "{*GUID*}" "*check-sum*"

Here, *filename* is the name of the file, *GUID* is the globally unique identifier associated with *filename,* and *check-sum* is a hexadecimal number that contains the checksum. This string must contain an even number of digits.

Assemblies and the internal Access Modifier

An integral part of C# programming is the assembly. An *assembly* is a file (or files) that contains all deployment and version information for a program. Assemblies are fundamental to the .NET environment. They provide mechanisms that support safe component interaction, interlanguage operability, and versioning. An assembly also defines a scope.

An assembly is composed of four sections. The first is the assembly *manifest.* The manifest contains information about the assembly, itself. This data includes such things as the name of the assembly, its version number, type mapping information, and cultural settings. The second section is *type metadata,* which is information about the data types used by the program. Among other benefits, type metadata aids in cross-language interoperability. The third part of an assembly is the *program code,* which is stored in Microsoft Intermediate Language (MSIL) format. The fourth constituent of an assembly is the resources used by the program.

Fortunately, when using C#, assemblies are produced automatically, with little or no extra effort on your part. The reason for this is that the **exe** file created when you compile a C# program is actually an assembly that contains your program's executable code as well as other types of information. Thus, when you compile a C# program, an assembly is automatically produced.

There are many other features and topics that relate to assemblies, but a discussion of these is outside the scope of this book. (Assemblies are an integral part of .NET development,

but are not technically a feature of the C# language.) However, there is one part of C# that relates directly to the assembly: the **internal** access modifier, which is examined next.

The internal Access Modifier

In addition to the access modifiers **public**, **private**, and **protected**, which you have been using throughout this book, C# also defines **internal**. The **internal** modifier declares that a member is known throughout all files in an assembly, but unknown outside that assembly. Thus, in simplified terms, a member marked as **internal** is known throughout a program, but not elsewhere. The **internal** access modifier is particularly useful when creating software components.

The **internal** modifier can be applied to classes and members of classes and to structures and members of structures. The **internal** modifier can also be applied to interface and enumeration declarations.

You can use **protected** in conjunction with **internal** to produce the **protected internal** access modifier pair. The **protected internal** access level can be given only to class members. A member declared with **protected internal** access is accessible within its own assembly or to derived types.

Here is an example that uses **internal**:

```
// Use internal.

using System;

class InternalTest {
  internal int x;
}

class InternalDemo {
  static void Main() {
    InternalTest ob = new InternalTest();

    ob.x = 10; // can access -- in same file

    Console.WriteLine("Here is ob.x: " + ob.x);

  }
}
```

Inside **InternalTest**, the field **x** is declared **internal**. This means that it is accessible within the program, as its use in **InternalDemo** shows, but unavailable outside the program.

17

CHAPTER

Runtime Type ID, Reflection, and Attributes

This chapter discusses three interrelated and powerful features: runtime type identification, reflection, and attributes. *Runtime type ID* is the mechanism that lets you identify a type during the execution of a program. *Reflection* is the feature that enables you to obtain information about a type. Using this information, you can construct and use objects at runtime. This feature is very powerful because it lets a program add functionality dynamically, during execution. An *attribute* describes a characteristic of some element of a C# program. For example, you can specify attributes for classes, methods, and fields, among others. Attributes can be interrogated at runtime, and the attribute information obtained. Attributes use both runtime type identification and reflection.

Runtime Type Identification

Runtime type identification (RTTI) allows the type of an object to be determined during program execution. RTTI is useful for many reasons. For example, you can discover precisely what type of object is being referred to by a base-class reference. Another use of RTTI is to test in advance whether a cast will succeed, preventing an invalid cast exception. Runtime type identification is also a key component of reflection.

C# includes three keywords that support runtime type identification: **is**, **as**, and **typeof**. Each is examined in turn.

Testing a Type with is

You can determine if an object is of a certain type by using the **is** operator. Its general form is shown here:

expr is *type*

Here, *expr* is an expression that describes an object whose type is being tested against *type.* If the type of *expr* is the same as, or compatible with, *type,* then the outcome of this operation is true. Otherwise, it is false. Thus, if the outcome is true, *expr* is some form of *type.* As it applies to **is**, one type is compatible with another if both are the same type, or if a reference, boxing, or unboxing conversion exists.

Here is an example that uses **is**:

```
// Demonstrate is.

using System;

class A {}
class B : A {}

class UseIs {
  static void Main() {
    A a = new A();
    B b = new B();

    if(a is A) Console.WriteLine("a is an A");
    if(b is A)
      Console.WriteLine("b is an A because it is derived from A");
    if(a is B)
      Console.WriteLine("This won't display -- a not derived from B");

    if(b is B) Console.WriteLine("B is a B");
    if(a is object) Console.WriteLine("a is an object");
  }
}
```

The output is shown here:

```
a is an A
b is an A because it is derived from A
B is a B
a is an object
```

Most of the **is** expressions are self-explanatory, but two may need a little discussion. First, notice this statement:

```
if(b is A)
  Console.WriteLine("b is an A because it is derived from A");
```

The **if** succeeds because **b** is an object of type **B**, which is derived from type **A**. Thus, **b** *is* an **A**. However, the reverse is not true. When this line is executed,

```
if(a is B)
  Console.WriteLine("This won't display -- a not derived from B");
```

the **if** does not succeed, because **a** is of type **A**, which is not derived from **B**. Thus, **a** *is not* **B**.

Using as

Sometimes you will want to try a conversion at runtime, but not throw an exception if the conversion fails (which is the case when a cast is used). To do this, use the **as** operator, which has this general form:

expr as *type*

Here, *expr* is the expression being converted to *type*. If the conversion succeeds, then a reference to *type* is returned. Otherwise, a null reference is returned. The **as** operator can be used to perform only reference, boxing, unboxing, or identity conversions.

The **as** operator offers a streamlined alternative to **is** in some cases. For example, consider the following program that uses **is** to prevent an invalid cast from occurring:

```
// Use is to avoid an invalid cast.

using System;

class A {}
class B : A {}

class CheckCast {
  static void Main() {
    A a = new A();
    B b = new B();

    // Check to see if a can be cast to B.
    if(a is B)  // if so, do the cast
      b = (B) a;
    else // if not, skip the cast
      b = null;

    if(b==null)
      Console.WriteLine("The cast in b = (B) a is NOT allowed.");
    else
      Console.WriteLine("The cast in b = (B) a is allowed");
  }
}
```

This program displays the following output:

```
The cast in b = (B) a is NOT allowed.
```

As the output shows, since **a** is not a **B**, the cast of **a** to **B** is invalid and is prevented by the **if** statement. However, this approach requires two steps. First, the validity of the cast must be confirmed. Second, the cast must be made. These two steps can be combined into one through the use of **as**, as the following program shows:

```
// Demonstrate as.

using System;

class A {}
class B : A {}

class CheckCast {
  static void Main() {
    A a = new A();
    B b = new B();

    b = a as B; // cast, if possible
```

```
    if(b==null)
      Console.WriteLine("The cast in b = (B) a is NOT allowed.");
    else
      Console.WriteLine("The cast in b = (B) a is allowed");
  }
}
```

Here is the output, which is the same as before:

```
The cast in b = (B) a is NOT allowed.
```

In this version, the **as** statement checks the validity of the cast and then, if valid, performs the cast, all in one statement.

Using typeof

Although useful in their own ways, the **as** and **is** operators simply test the compatibility of two types. Often, you will need to obtain information about a type. To do this, C# supplies the **typeof** operator. It retrieves a **System.Type** object for a given type. Using this object, you can determine the type's characteristics.

The **typeof** operator has this general form:

typeof(*type*)

Here, *type* is the type being obtained. The **Type** object returned encapsulates the information associated with *type.*

Once you have obtained a **Type** object for a given type, you can obtain information about it through the use of various properties, fields, and methods defined by **Type**. **Type** is a large class with many members, and a discussion is deferred until the next section, where reflection is examined. However, to briefly demonstrate **Type**, the following program uses three of its properties: **FullName**, **IsClass**, and **IsAbstract**. To obtain the full name of the type, use **FullName**. **IsClass** returns true if the type is a class. **IsAbstract** returns true if a class is abstract.

```
// Demonstrate typeof.

using System;
using System.IO;

class UseTypeof {
  static void Main() {
    Type t = typeof(StreamReader);

    Console.WriteLine(t.FullName);

    if(t.IsClass) Console.WriteLine("Is a class.");
    if(t.IsAbstract) Console.WriteLine("Is abstract.");
    else Console.WriteLine("Is concrete.");
  }
}
```

This program outputs the following:

```
System.IO.StreamReader
Is a class.
Is concrete.
```

This program obtains a **Type** object that describes **StreamReader**. It then displays the full name, and determines if it is a class and whether it is abstract.

Reflection

Reflection is the feature that enables you to obtain information about a type. The term *reflection* comes from the way the process works: A **Type** object mirrors the underlying type that it represents. To obtain information, you ask the **Type** object questions, and it returns (reflects) the information associated with the type back to you. Reflection is a powerful mechanism because it allows you to learn and use the capabilities of types that are known only at runtime.

Many of the classes that support reflection are part of the .NET Reflection API, which is in the **System.Reflection** namespace. Thus, you will normally include the following in programs that use reflection:

```
using System.Reflection;
```

The Reflection Core: System.Type

System.Type is at the core of the reflection subsystem because it encapsulates a type. It contains many properties and methods that you will use to obtain information about a type at runtime. **Type** is derived from an abstract class called **System.Reflection.MemberInfo**.

MemberInfo defines the following read-only properties:

Property	Description
Type DeclaringType	Obtains the type of the class or interface in which the member is declared.
MemberTypes MemberType	Obtains the kind of the member. This value indicates if the member is a field, method, property, event, or constructor.
int MetadataToken	Obtains a value associated with a specific metadata.
Module Module	Obtains a **Module** object that represents the module (an executable file) in which the reflected type resides.
string Name	The name of the type.
Type ReflectedType	The type of the object being reflected.

Notice that the return type of **MemberType** is **MemberTypes**. **MemberTypes** is an enumeration that defines values that indicate the various member types. Among others, these include

MemberTypes.Constructor
MemberTypes.Method
MemberTypes.Field
MemberTypes.Event
MemberTypes.Property

Thus, the type of a member can be determined by checking **MemberType**. For example, if **MemberType** equals **MemberTypes.Method**, then that member is a method.

MemberInfo includes two abstract methods: **GetCustomAttributes( )** and **IsDefined( )**. These both relate to attributes. The first obtains a list of the custom attributes associated with the invoking object. The second determines if an attribute is defined for the invoking object. (Attributes are described later in this chapter.)

To the methods and properties defined by **MemberInfo**, **Type** adds a great many of its own. For example, here are several commonly used methods defined by **Type**:

Method	Purpose
ConstructorInfo[ ] GetConstructors( )	Obtains a list of the constructors for the specified type.
EventInfo[ ] GetEvents( )	Obtains a list of events for the specified type.
FieldInfo[ ] GetFields( )	Obtains a list of the fields for the specified type.
Type[ ] GetGenericArguments( )	Obtains a list of the type arguments bound to a closed constructed generic type or the type parameters if the specified type is a generic type definition. For an open constructed type, the list may contain both type arguments and type parameters. (See Chapter 18 for a discussion of generics.)
MemberInfo[ ] GetMembers( )	Obtains a list of the members for the specified type.
MethodInfo[ ] GetMethods( )	Obtains a list of methods for the specified type.
PropertyInfo[ ] GetProperties( )	Obtains a list of properties for the specified type.

Here are several commonly used, read-only properties defined by **Type**:

Property	Purpose
Assembly Assembly	Obtains the assembly for the specified type.
TypeAttributes Attributes	Obtains the attributes for the specified type.
Type BaseType	Obtains the immediate base type for the specified type.
string FullName	Obtains the complete name of the specified type.
bool IsAbstract	Is true if the specified type is abstract.
bool isArray	Is true if the specified type is an array.
bool IsClass	Is true if the specified type is a class.
bool IsEnum	Is true if the specified type is an enumeration.
bool IsGenericParameter	Is true if the specified type is a generic type parameter. (See Chapter 18 for a discussion of generics.)
bool IsGenericType	Is true if the specified type is a generic type. (See Chapter 18 for a discussion of generics.)
string Namespace	Obtains the namespace of the specified type.

Using Reflection

Using **Type**'s methods and properties, it is possible to obtain detailed information about a type at runtime. This is an extremely powerful feature, because once you have obtained information about a type, you can invoke its constructors, call its methods, and use its properties. Thus, reflection enables you to use code that was not available at compile time.

The Reflection API is quite large, and it is not possible to cover the entire topic here. (Complete coverage of reflection could easily fill an entire book!) However, because the Reflection API is logically designed, once you understand how to use a part of it, the rest just falls into place. With this thought in mind, the following sections demonstrate four key reflection techniques: obtaining information about methods, invoking methods, constructing objects, and loading types from assemblies.

Obtaining Information About Methods

Once you have a **Type** object, you can obtain a list of methods supported by the type by using **GetMethods( )**. One form is shown here:

```
MethodInfo[ ] GetMethods( )
```

It returns an array of **MethodInfo** objects that describe the methods supported by the invoking type. **MethodInfo** is in the **System.Reflection** namespace.

MethodInfo is derived from the abstract class **MethodBase**, which inherits **MemberInfo**. Thus, the properties and methods defined by all three of these classes are available for your use. For example, to obtain the name of a method, use the **Name** property. Two members that are of particular interest at this time are **ReturnType** and **GetParameters( )**.

The return type of a method is found in the read-only **ReturnType** property, which is an object of **Type**.

The method **GetParameters( )** returns a list of the parameters associated with a method. It has this general form:

```
ParameterInfo[ ] GetParameters( );
```

The parameter information is held in a **ParameterInfo** object. **ParameterInfo** defines a large number of properties and methods that describe the parameter. Two properties that are of particular value are **Name**, which is a string that contains the name of the parameter, and **ParameterType**, which describes the parameter's type. The parameter's type is encapsulated within a **Type** object.

Here is a program that uses reflection to obtain the methods supported by a class called **MyClass**. For each method, it displays the return type and name of the method, and the name and type of any parameters that each method may have.

```
// Analyze methods using reflection.

using System;
using System.Reflection;

class MyClass {
  int x;
  int y;
```

```
  public MyClass(int i, int j) {
    x = i;
    y = j;
  }

  public int Sum() {
    return x+y;
  }

  public bool IsBetween(int i) {
    if(x < i && i < y) return true;
    else return false;
  }

  public void Set(int a, int b) {
    x = a;
    y = b;
  }

  public void Set(double a, double b) {
    x = (int) a;
    y = (int) b;
  }

  public void Show() {
    Console.WriteLine(" x: {0}, y: {1}", x, y);
  }
}

class ReflectDemo {
  static void Main() {
    Type t = typeof(MyClass); // get a Type object representing MyClass

    Console.WriteLine("Analyzing methods in " + t.Name);
    Console.WriteLine();

    Console.WriteLine("Methods supported: ");

    MethodInfo[] mi = t.GetMethods();

    // Display methods supported by MyClass.
    foreach(MethodInfo m in mi) {
      // Display return type and name.
      Console.Write("   " + m.ReturnType.Name +
                    " " + m.Name + "(");

      // Display parameters.
      ParameterInfo[] pi = m.GetParameters();

      for(int i=0; i < pi.Length; i++) {
        Console.Write(pi[i].ParameterType.Name +
                      " " + pi[i].Name);
        if(i+1 < pi.Length) Console.Write(", ");
      }
```

```
      Console.WriteLine(")");

      Console.WriteLine();
    }
  }
}
```

The output is shown here:

```
Analyzing methods in MyClass

Methods supported:
   Int32 Sum()

   Boolean IsBetween(Int32 i)

   Void Set(Int32 a, Int32 b)

   Void Set(Double a, Double b)

   Void Show()

   Type GetType()

   String ToString()

   Boolean Equals(Object obj)

   Int32 GetHashCode()
```

Notice that in addition to the methods defined by **MyClass**, the methods defined by **object** are also displayed. This is because all types in C# inherit **object**. Also notice that the .NET structure names are used for the type names. Observe that **Set( )** is displayed twice. This is because **Set( )** is overloaded. One version takes **int** arguments. The other takes **double** arguments.

Let's look at this program closely. First, notice that **MyClass** defines a public constructor and a number of public methods, including the overloaded **Set( )** method.

Inside **Main( )**, a **Type** object representing **MyClass** is obtained using this line of code:

```
Type t = typeof(MyClass); // get a Type object representing MyClass
```

Recall that **typeof** returns a **Type** object that represents the specified type, which in this case is **MyClass**.

Using **t** and the Reflection API, the program then displays information about the methods supported by **MyClass**. First, a list of the methods is obtained by the following statement:

```
MethodInfo[] mi = t.GetMethods();
```

Next, a **foreach** loop is established that cycles through **mi**. With each pass, the return type, name, and parameters for each method are displayed by the following code:

```
// Display return type and name.
Console.Write("   " + m.ReturnType.Name +
              " " + m.Name + "(");
```

```
// Display parameters.
ParameterInfo[] pi = m.GetParameters();

for(int i=0; i < pi.Length; i++) {
  Console.Write(pi[i].ParameterType.Name +
                " " + pi[i].Name);
  if(i+1 < pi.Length) Console.Write(", ");
}
```

In this sequence, the parameters associated with each method are obtained by calling **GetParameters( )** and stored in the **pi** array. Then a **for** loop cycles through the **pi** array, displaying the type and name of each parameter. The key point is that this information is obtained dynamically at runtime without relying on prior knowledge of **MyClass**.

A Second Form of GetMethods( )

A second form of **GetMethods( )** lets you specify various flags that filter the methods that are retrieved. It has this general form:

MethodInfo[ ] GetMethods(BindingFlags *flags*)

This version obtains only those methods that match the criteria you specify. **BindingFlags** is an enumeration. Here are several commonly used values:

Value	Meaning
DeclaredOnly	Retrieves only those methods defined by the specified class. Inherited methods are not included.
Instance	Retrieves instance methods.
NonPublic	Retrieves nonpublic methods.
Public	Retrieves public methods.
Static	Retrieves **static** methods.

You can OR together two or more flags. In fact, minimally you must include either **Instance** or **Static** with **Public** or **NonPublic**. Failure to do so will result in no methods being retrieved.

One of the main uses of the **BindingFlags** form of **GetMethods( )** is to enable you to obtain a list of the methods defined by a class without also retrieving the inherited methods. This is especially useful for preventing the methods defined by **object** from being obtained. For example, try substituting this call to **GetMethods( )** into the preceding program:

```
// Now, only methods declared by MyClass are obtained.
MethodInfo[] mi = t.GetMethods(BindingFlags.DeclaredOnly |
                               BindingFlags.Instance |
                               BindingFlags.Public) ;
```

After making this change, the program produces the following output:

```
Analyzing methods in MyClass

Methods supported:
   Int32 Sum()
```

```
Boolean IsBetween(Int32 i)

Void Set(Int32 a, Int32 b)

Void Set(Double a, Double b)

Void Show()
```

As you can see, only those methods explicitly defined by **MyClass** are displayed.

Calling Methods Using Reflection

Once you know what methods a type supports, you can call one or more of them. To do this, you will use the **Invoke( )** method that is contained in **MethodInfo**. One of its forms is shown here:

object Invoke(object *ob*, object[ ] *args*)

Here, *ob* is a reference to the object on which the method is invoked. For **static** methods, pass **null** to *ob*. Any arguments that need to be passed to the method are specified in the array *args*. If no arguments are needed, *args* must be **null**. Also, *args* must contain exactly the same number of elements as there are arguments. Therefore, if two arguments are needed, then *args* must be two elements long. It can't, for example, be three or four elements long. The value returned by the invoked method is returned by **Invoke( )**.

To call a method, simply call **Invoke( )** on an instance of **MethodInfo** that was obtained by calling **GetMethods( )**. The following program demonstrates the procedure:

```
// Invoke methods using reflection.

using System;
using System.Reflection;

class MyClass {
  int x;
  int y;

  public MyClass(int i, int j) {
    x = i;
    y = j;
  }

  public int Sum() {
    return x+y;
  }

  public bool IsBetween(int i) {
    if((x < i) && (i < y)) return true;
    else return false;
  }

  public void Set(int a, int b) {
    Console.Write("Inside Set(int, int). ");
    x = a;
```

```
    y = b;
    Show();
  }

  // Overload set.
  public void Set(double a, double b) {
    Console.Write("Inside Set(double, double). ");
    x = (int) a;
    y = (int) b;
    Show();
  }

  public void Show() {
    Console.WriteLine("Values are x: {0}, y: {1}", x, y);
  }
}

class InvokeMethDemo {
  static void Main() {
    Type t = typeof(MyClass);
    MyClass reflectOb = new MyClass(10, 20);
    int val;

    Console.WriteLine("Invoking methods in " + t.Name);
    Console.WriteLine();
    MethodInfo[] mi = t.GetMethods();

    // Invoke each method.
    foreach(MethodInfo m in mi) {
      // Get the parameters.
      ParameterInfo[] pi = m.GetParameters();

      if(m.Name.CompareTo("Set")==0 &&
         pi[0].ParameterType == typeof(int)) {
        object[] args = new object[2];
        args[0] = 9;
        args[1] = 18;
        m.Invoke(reflectOb, args);
      }
      else if(m.Name.CompareTo("Set")==0 &&
              pi[0].ParameterType == typeof(double)) {
        object[] args = new object[2];
        args[0] = 1.12;
        args[1] = 23.4;
        m.Invoke(reflectOb, args);
      }
      else if(m.Name.CompareTo("Sum")==0) {
        val = (int) m.Invoke(reflectOb, null);
        Console.WriteLine("sum is " + val);
      }
      else if(m.Name.CompareTo("IsBetween")==0) {
        object[] args = new object[1];
        args[0] = 14;
        if((bool) m.Invoke(reflectOb, args))
```

```
          Console.WriteLine("14 is between x and y");
      }
      else if(m.Name.CompareTo("Show")==0) {
        m.Invoke(reflectOb, null);
      }
    }
  }
}
```

The output is shown here:

```
Invoking methods in MyClass

sum is 30
14 is between x and y
Inside Set(int, int). Values are x: 9, y: 18
Inside Set(double, double). Values are x: 1, y: 23
Values are x: 1, y: 23
```

Look closely at how the methods are invoked. First, a list of methods is obtained. Then, inside the **foreach** loop, parameter information is retrieved. Next, using a series of if/else statements, each method is executed with the proper type and number of arguments. Pay special attention to the way that the overloaded **Set( )** method is executed by the following code:

```
if(m.Name.CompareTo("Set")==0 &&
   pi[0].ParameterType == typeof(int)) {
  object[] args = new object[2];
  args[0] = 9;
  args[1] = 18;
  m.Invoke(reflectOb, args);
}
else if(m.Name.CompareTo("Set")==0 &&
  pi[0].ParameterType == typeof(double)) {
  object[] args = new object[2];
  args[0] = 1.12;
  args[1] = 23.4;
  m.Invoke(reflectOb, args);
}
```

If the name of the method is **Set**, then the type of the first parameter is tested to determine which version of the method was found. If it was **Set(int, int)**, then **int** arguments are loaded into **args**. Otherwise, **double** arguments are used.

Obtaining a Type's Constructors

In the previous example, there is no advantage to using reflection to invoke methods on **MyClass** since an object of type **MyClass** was explicitly created. It would be easier to just call its methods normally. However, the power of reflection starts to become apparent when an object is created dynamically at runtime. To do this, you will need to first obtain a list of the constructors. Then, you will create an instance of the type by invoking one of the constructors. This mechanism allows you to instantiate any type of object at runtime without naming it in a declaration statement.

To obtain the constructors for a type, call **GetConstructors( )** on a **Type** object. One commonly used form is shown here:

ConstructorInfo[ ] GetConstructors( )

It returns an array of **ConstructorInfo** objects that describe the constructors.

ConstructorInfo is derived from the abstract class **MethodBase**, which inherits **MemberInfo**. It also defines several members of its own. The one we are interested in is **GetParameters( )**, which returns a list of the parameters associated with a constructor. It works just like **GetParameters( )** defined by **MethodInfo**, described earlier.

Once an appropriate constructor has been found, an object is created by calling the **Invoke( )** method defined by **ConstructorInfo**. One form is shown here:

object Invoke(object[ ] *args*)

Any arguments that need to be passed to the method are specified in the array *args*. If no arguments are needed, pass **null** to *args*. In all cases, *args* must contain exactly the same number of elements as there are arguments and the types of arguments must be compatible with the types of the parameters. **Invoke( )** returns a reference to the object that was constructed.

The following program uses reflection to create an instance of **MyClass**:

```
// Create an object using reflection.

using System;
using System.Reflection;

class MyClass {
  int x;
  int y;

  public MyClass(int i) {
    Console.WriteLine("Constructing MyClass(int, int). ");
    x = y = i;
  }

  public MyClass(int i, int j) {
    Console.WriteLine("Constructing MyClass(int, int). ");
    x = i;
    y = j;
    Show();
  }

  public int Sum() {
    return x+y;
  }

  public bool IsBetween(int i) {
    if((x < i) && (i < y)) return true;
    else return false;
  }

  public void Set(int a, int b) {
    Console.Write("Inside Set(int, int). ");
```

```
    x = a;
    y = b;
    Show();
  }

  // Overload Set.
  public void Set(double a, double b) {
    Console.Write("Inside Set(double, double). ");
    x = (int) a;
    y = (int) b;
    Show();
  }

  public void Show() {
    Console.WriteLine("Values are x: {0}, y: {1}", x, y);
  }

}

class InvokeConsDemo {
  static void Main() {
    Type t = typeof(MyClass);
    int val;

    // Get constructor info.
    ConstructorInfo[] ci = t.GetConstructors();

    Console.WriteLine("Available constructors: ");
    foreach(ConstructorInfo c in ci) {
      // Display return type and name.
      Console.Write("   " + t.Name + "(");

      // Display parameters.
      ParameterInfo[] pi = c.GetParameters();

      for(int i=0; i < pi.Length; i++) {
        Console.Write(pi[i].ParameterType.Name +
                      " " + pi[i].Name);
        if(i+1 < pi.Length) Console.Write(", ");
      }

      Console.WriteLine(")");
    }
    Console.WriteLine();

    // Find matching constructor.
    int x;

    for(x=0; x < ci.Length; x++) {
      ParameterInfo[] pi =  ci[x].GetParameters();
      if(pi.Length == 2) break;
    }

    if(x == ci.Length) {
```

```
      Console.WriteLine("No matching constructor found.");
      return;
    }
    else
      Console.WriteLine("Two-parameter constructor found.\n");

    // Construct the object.
    object[] consargs = new object[2];
    consargs[0] = 10;
    consargs[1] = 20;
    object reflectOb = ci[x].Invoke(consargs);

    Console.WriteLine("\nInvoking methods on reflectOb.");
    Console.WriteLine();
    MethodInfo[] mi = t.GetMethods();

    // Invoke each method.
    foreach(MethodInfo m in mi) {
      // Get the parameters.
      ParameterInfo[] pi = m.GetParameters();

      if(m.Name.CompareTo("Set")==0 &&
         pi[0].ParameterType == typeof(int)) {
        // This is Set(int, int).
        object[] args = new object[2];
        args[0] = 9;
        args[1] = 18;
        m.Invoke(reflectOb, args);
      }
      else if(m.Name.CompareTo("Set")==0 &&
              pi[0].ParameterType == typeof(double)) {
        // This is Set(double, double).
        object[] args = new object[2];
        args[0] = 1.12;
        args[1] = 23.4;
        m.Invoke(reflectOb, args);
      }
      else if(m.Name.CompareTo("Sum")==0) {
        val = (int) m.Invoke(reflectOb, null);
        Console.WriteLine("sum is " + val);
      }
      else if(m.Name.CompareTo("IsBetween")==0) {
        object[] args = new object[1];
        args[0] = 14;
        if((bool) m.Invoke(reflectOb, args))
          Console.WriteLine("14 is between x and y");
      }
      else if(m.Name.CompareTo("Show")==0) {
        m.Invoke(reflectOb, null);
      }
    }
  }
}
```

The output is shown here:

```
Available constructors:
   MyClass(Int32 i)
   MyClass(Int32 i, Int32 j)

Two-parameter constructor found.

Constructing MyClass(int, int).
Values are x: 10, y: 20

Invoking methods on reflectOb.

sum is 30
14 is between x and y
Inside Set(int, int). Values are x: 9, y: 18
Inside Set(double, double). Values are x: 1, y: 23
Values are x: 1, y: 23
```

Let's look at how reflection is used to construct a **MyClass** object. First, a list of the public constructors is obtained using the following statement:

```
ConstructorInfo[] ci = t.GetConstructors();
```

Next, for the sake of illustration, the constructors are displayed. Then the list is searched for a constructor that takes two arguments, using this code:

```
for(x=0; x < ci.Length; x++) {
  ParameterInfo[] pi =  ci[x].GetParameters();
  if(pi.Length == 2) break;
}
```

If the constructor is found (as it will be in this case), an object is instantiated by the following sequence:

```
// Construct the object.
object[] consargs = new object[2];
consargs[0] = 10;
consargs[1] = 20;
object reflectOb = ci[x].Invoke(consargs);
```

After the call to **Invoke( )**, **reflectOb** will refer to an object of type **MyClass**. The program then executes methods on that instance.

One important point needs to be made. In this example, for the sake of simplicity, it was assumed that the only two-argument constructor was one that took two **int** arguments. Obviously, in real-world code this would need to be verified by checking the parameter type of each argument.

Obtaining Types from Assemblies

In the preceding example, everything about **MyClass** has been discovered using reflection except for one item: the type **MyClass**, itself. That is, although the preceding examples dynamically determined information about **MyClass**, they still relied upon the fact that the

type name **MyClass** was known in advance and used in a **typeof** statement to obtain a **Type** object upon which all of the reflection methods either directly or indirectly operated. Although this might be useful in a number of circumstances, the full power of reflection is found when the types available to a program are determined dynamically by analyzing the contents of other assemblies.

As you know from Chapter 16, an assembly carries with it type information about the classes, structures, and so on, that it contains. The Reflection API allows you to load an assembly, discover information about it, and create instances of any of its publicly available types. Using this mechanism, a program can search its environment, utilizing functionality that might be available without having to explicitly define that functionality at compile time. This is an extremely potent, and exciting, concept. For example, you can imagine a program that acts as a "type browser," displaying the types available on a system. Another application could be a design tool that lets you visually "wire together" a program that is composed of the various types supported by the system. Since all information about a type is discoverable, there is no inherent limitation to the ways reflection can be applied.

To obtain information about an assembly, you will first create an **Assembly** object. The **Assembly** class does not define a public constructor. Instead, an **Assembly** object is obtained by calling one of its methods. The one we will use is **LoadFrom( )**, which loads an assembly given its filename. The form we will use is shown here:

static Assembly LoadFrom(string *filename*)

Here, *filename* specifies the filename of the assembly.

Once you have obtained an **Assembly** object, you can discover the types that it defines by calling **GetTypes( )** on it. Here is its general form:

Type[ ] GetTypes( )

It returns an array of the types contained in the assembly.

To demonstrate the discovery of types in an assembly, you will need two files. The first will contain a set of classes that will be discovered by the second. To begin, create a file called **MyClasses.cs** that contains the following:

```
// A file that contains three classes. Call this file MyClasses.cs.

using System;

class MyClass {
  int x;
  int y;

  public MyClass(int i) {
    Console.WriteLine("Constructing MyClass(int). ");
    x = y = i;
    Show();
  }

  public MyClass(int i, int j) {
    Console.WriteLine("Constructing MyClass(int, int). ");
    x = i;
    y = j;
```

```
    Show();
  }

  public int Sum() {
    return x+y;
  }

  public bool IsBetween(int i) {
    if((x < i) && (i < y)) return true;
    else return false;
  }

  public void Set(int a, int b) {
    Console.Write("Inside Set(int, int). ");
    x = a;
    y = b;
    Show();
  }

  // Overload Set.
  public void Set(double a, double b) {
    Console.Write("Inside Set(double, double). ");
    x = (int) a;
    y = (int) b;
    Show();
  }

  public void Show() {
    Console.WriteLine("Values are x: {0}, y: {1}", x, y);
  }
}

class AnotherClass {
  string msg;

  public AnotherClass(string str) {
    msg = str;
  }

  public void Show() {
    Console.WriteLine(msg);
  }
}

class Demo {
  static void Main() {
    Console.WriteLine("This is a placeholder.");
  }
}
```

This file contains **MyClass**, which we have been using in the previous examples. It also adds a second class called **AnotherClass** and a third class called **Demo**. Thus, the assembly

produced by this program will contain three classes. Next, compile this file so the file **MyClasses.exe** is produced. This is the assembly that will be interrogated.

The program that will discover information about **MyClasses.exe** is shown here. Enter it at this time.

```
/* Locate an assembly, determine types, and create
   an object using reflection. */

using System;
using System.Reflection;

class ReflectAssemblyDemo {
  static void Main() {
    int val;

    // Load the MyClasses.exe assembly.
    Assembly asm = Assembly.LoadFrom("MyClasses.exe");

    // Discover what types MyClasses.exe contains.
    Type[] alltypes = asm.GetTypes();
    foreach(Type temp in alltypes)
      Console.WriteLine("Found: " + temp.Name);

    Console.WriteLine();

    // Use the first type, which is MyClass in this case.
    Type t = alltypes[0]; // use first class found
    Console.WriteLine("Using: " + t.Name);

    // Obtain constructor info.
    ConstructorInfo[] ci = t.GetConstructors();

    Console.WriteLine("Available constructors: ");
    foreach(ConstructorInfo c in ci) {
      // Display return type and name.
      Console.Write("   " + t.Name + "(");

      // Display parameters.
      ParameterInfo[] pi = c.GetParameters();

      for(int i=0; i < pi.Length; i++) {
        Console.Write(pi[i].ParameterType.Name +
                      " " + pi[i].Name);
        if(i+1 < pi.Length) Console.Write(", ");
      }

      Console.WriteLine(")");
    }
    Console.WriteLine();

    // Find matching constructor.
    int x;

    for(x=0; x < ci.Length; x++) {
```

```
  ParameterInfo[] pi =  ci[x].GetParameters();
  if(pi.Length == 2) break;
}

if(x == ci.Length) {
  Console.WriteLine("No matching constructor found.");
  return;
}
else
  Console.WriteLine("Two-parameter constructor found.\n");

// Construct the object.
object[] consargs = new object[2];
consargs[0] = 10;
consargs[1] = 20;
object reflectOb = ci[x].Invoke(consargs);

Console.WriteLine("\nInvoking methods on reflectOb.");
Console.WriteLine();
MethodInfo[] mi = t.GetMethods();

// Invoke each method.
foreach(MethodInfo m in mi) {
  // Get the parameters.
  ParameterInfo[] pi = m.GetParameters();

  if(m.Name.CompareTo("Set")==0 &&
     pi[0].ParameterType == typeof(int)) {
    // This is Set(int, int).
    object[] args = new object[2];
    args[0] = 9;
    args[1] = 18;
    m.Invoke(reflectOb, args);
  }
  else if(m.Name.CompareTo("Set")==0 &&
     pi[0].ParameterType == typeof(double)) {
    // This is Set(double, double).
    object[] args = new object[2];
    args[0] = 1.12;
    args[1] = 23.4;
    m.Invoke(reflectOb, args);
  }
  else if(m.Name.CompareTo("Sum")==0) {
    val = (int) m.Invoke(reflectOb, null);
    Console.WriteLine("sum is " + val);
  }
  else if(m.Name.CompareTo("IsBetween")==0) {
    object[] args = new object[1];
    args[0] = 14;
    if((bool) m.Invoke(reflectOb, args))
      Console.WriteLine("14 is between x and y");
  }
  else if(m.Name.CompareTo("Show")==0) {
    m.Invoke(reflectOb, null);
```

```
      }
    }

  }
}
```

The output from the program is shown here:

```
Found: MyClass
Found: AnotherClass
Found: Demo

Using: MyClass
Available constructors:
   MyClass(Int32 i)
   MyClass(Int32 i, Int32 j)

Two-parameter constructor found.

Constructing MyClass(int, int).
Values are x: 10, y: 20

Invoking methods on reflectOb.

sum is 30
14 is between x and y
Inside Set(int, int). Values are x: 9, y: 18
Inside Set(double, double). Values are x: 1, y: 23
Values are x: 1, y: 23
```

As the output shows, all three classes contained within **MyClasses.exe** were found. The first one, which in this case was **MyClass**, was then used to instantiate an object and execute methods.

The types in **MyClasses.exe** are discovered using this sequence of code, which is near the start of **Main( )**:

```
// Load the MyClasses.exe assembly.
Assembly asm = Assembly.LoadFrom("MyClasses.exe");

// Discover what types MyClasses.exe contains.
Type[] alltypes = asm.GetTypes();
foreach(Type temp in alltypes)
  Console.WriteLine("Found: " + temp.Name);
```

You can use such a sequence whenever you need to dynamically load and interrogate an assembly.

On a related point, an assembly need not be an **exe** file. Assemblies can also be contained in dynamic link library (DLL) files that use the **dll** extension. For example, if you were to compile **MyClasses.cs** using this command line,

```
csc /t:library MyClasses.cs
```

then the output file would be **MyClasses.dll**. One advantage to putting code into a DLL is that no **Main( )** method is required. All **exe** files require an entry point, such as **Main( )**, that

defines where execution begins. This is why the **Demo** class contained a placeholder **Main( )** method. Such a method is not required by a DLL. If you try making **MyClass** into a DLL, you will need to change the call to **LoadFrom( )** as shown here:

```
Assembly asm = Assembly.LoadFrom("MyClasses.dll");
```

Fully Automating Type Discovery

Before we leave the topic of reflection, one last example will be instructive. Even though the preceding program was able to fully use **MyClass** without explicitly specifying **MyClass** in the program, it still relied upon prior knowledge of the contents of **MyClass**. For example, the program knew the names of its methods, such as **Set** and **Sum**. However, using reflection it is possible to utilize a type about which you have no prior knowledge. To do this, you must discover all information necessary to construct an object and to generate method calls. Such an approach would be useful to a visual design tool, for example, because it could utilize the types available on the system.

To see how the full dynamic discovery of a type can be accomplished, consider the following example, which loads the **MyClasses.exe** assembly, constructs a **MyClass** object, and then calls all of the methods declared by **MyClass**, all without assuming any prior knowledge:

```
// Utilize MyClass without assuming any prior knowledge.

using System;
using System.Reflection;

class ReflectAssemblyDemo {
  static void Main() {
    int val;
    Assembly asm = Assembly.LoadFrom("MyClasses.exe");

    Type[] alltypes = asm.GetTypes();

    Type t = alltypes[0]; // use first class found

    Console.WriteLine("Using: " + t.Name);

    ConstructorInfo[] ci = t.GetConstructors();

    // Use first constructor found.
    ParameterInfo[] cpi = ci[0].GetParameters();
    object reflectOb;

    if(cpi.Length > 0) {
      object[] consargs = new object[cpi.Length];

      // Initialize args.
      for(int n=0; n < cpi.Length; n++)
        consargs[n] = 10 + n * 20;

      // Construct the object.
      reflectOb = ci[0].Invoke(consargs);
    } else
```

```
    reflectOb = ci[0].Invoke(null);


  Console.WriteLine("\nInvoking methods on reflectOb.");
  Console.WriteLine();

  // Ignore inherited methods.
  MethodInfo[] mi = t.GetMethods(BindingFlags.DeclaredOnly |
                                 BindingFlags.Instance |
                                 BindingFlags.Public) ;

  // Invoke each method.
  foreach(MethodInfo m in mi) {
    Console.WriteLine("Calling {0} ", m.Name);

    // Get the parameters.
    ParameterInfo[] pi = m.GetParameters();

    // Execute methods.
    switch(pi.Length) {
      case 0: // no args
        if(m.ReturnType == typeof(int)) {
          val = (int) m.Invoke(reflectOb, null);
          Console.WriteLine("Result is " + val);
        }
        else if(m.ReturnType == typeof(void)) {
          m.Invoke(reflectOb, null);
        }
        break;
      case 1: // one arg
        if(pi[0].ParameterType == typeof(int)) {
          object[] args = new object[1];
          args[0] = 14;
          if((bool) m.Invoke(reflectOb, args))
            Console.WriteLine("14 is between x and y");
          else
            Console.WriteLine("14 is not between x and y");
        }
        break;
      case 2: // two args
        if((pi[0].ParameterType == typeof(int)) &&
           (pi[1].ParameterType == typeof(int))) {
          object[] args = new object[2];
          args[0] = 9;
          args[1] = 18;
          m.Invoke(reflectOb, args);
        }
        else if((pi[0].ParameterType == typeof(double)) &&
                (pi[1].ParameterType == typeof(double))) {
          object[] args = new object[2];
          args[0] = 1.12;
          args[1] = 23.4;
          m.Invoke(reflectOb, args);
        }
        break;
```

```
      }
      Console.WriteLine();
    }

  }
}
```

Here is the output produced by the program:

```
Using: MyClass
Constructing MyClass(int).
Values are x: 10, y: 10

Invoking methods on reflectOb.

Calling Sum
Result is 20

Calling IsBetween
14 is not between x and y

Calling Set
Inside Set(int, int). Values are x: 9, y: 18

Calling Set
Inside Set(double, double). Values are x: 1, y: 23

Calling Show
Values are x: 1, y: 23
```

The operation of the program is straightforward, but a couple of points are worth mentioning. First, notice that only the methods explicitly declared by **MyClass** are obtained and used. This is accomplished by using the **BindingFlags** form of **GetMethods( )**. The reason for this is to prevent calling the methods inherited from **object**. Second, notice how the number of parameters and return type of each method are obtained dynamically. A **switch** statement determines the number of parameters. Within each **case**, the parameter type(s) and return type are checked. A method call is then constructed based on this information.

Attributes

C# allows you to add declarative information to a program in the form of an *attribute.* An attribute defines additional information (metadata) that is associated with a class, structure, method, and so on. For example, you might define an attribute that determines the type of button that a class will display. Attributes are specified between square brackets, preceding the item to which they apply. Thus, an attribute is not a member of a class. Rather, an attribute specifies supplemental information that is attached to an item.

Attribute Basics

An attribute is supported by a class that inherits **System.Attribute**. Thus, all attribute classes must be subclasses of **Attribute**. Although **Attribute** defines substantial functionality, this functionality is not always needed when working with attributes. By convention, attribute classes often use the suffix **Attribute**. For example, **ErrorAttribute** would be a name for an attribute class that described an error.

When an attribute class is declared, it is preceded by an attribute called **AttributeUsage**. This built-in attribute specifies the types of items to which the attribute can be applied. Thus, the usage of an attribute can be restricted to methods, for example.

Creating an Attribute

In an attribute class, you will define the members that support the attribute. Often attribute classes are quite simple, containing just a small number of fields or properties. For example, an attribute might define a remark that describes the item to which the attribute is being attached. Such an attribute might look like this:

```
 [AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute {
  string pri_remark; // underlies Remark property

  public RemarkAttribute(string comment) {
    pri_remark = comment;
  }

  public string Remark {
    get {
      return pri_remark;
    }
  }
}
```

Let's look at this class, line by line.

The name of this attribute is **RemarkAttribute**. Its declaration is preceded by the **AttributeUsage** attribute, which specifies that **RemarkAttribute** can be applied to all types of items. Using **AttributeUsage**, it is possible to narrow the list of items to which an attribute can be attached, and we will examine its capabilities later in this chapter.

Next, **RemarkAttribute** is declared and it inherits **Attribute**. Inside **RemarkAttribute** there is one private field, **pri_remark**, which supports one public, read-only property: **Remark**. This property holds the description that will be associated with the attribute. (**Remark** could also have been declared as an auto-implemented property with a private set accessor, but a read-only property is used for the purposes of illustration.) There is one public constructor that takes a string argument and assigns it to **Remark**.

At this point, no other steps are needed, and **RemarkAttribute** is ready for use.

Attaching an Attribute

Once you have defined an attribute class, you can attach the attribute to an item. An attribute precedes the item to which it is attached and is specified by enclosing its constructor inside square brackets. For example, here is how **RemarkAttribute** can be associated with a class:

```
[RemarkAttribute("This class uses an attribute.")]
class UseAttrib {
  // ...
}
```

This constructs a **RemarkAttribute** that contains the comment, "This class uses an attribute." This attribute is then associated with **UseAttrib**.

When attaching an attribute, it is not actually necessary to specify the **Attribute** suffix. For example, the preceding class could be declared this way:

```
[Remark("This class uses an attribute.")]
class UseAttrib {
  // ...
}
```

Here, only the name **Remark** is used. Although the short form is correct, it is usually safer to use the full name when attaching attributes, because it avoids possible confusion and ambiguity.

Obtaining an Object's Attributes

Once an attribute has been attached to an item, other parts of the program can retrieve the attribute. To retrieve an attribute, you will usually use one of two methods. The first is **GetCustomAttributes( )**, which is defined by **MemberInfo** and inherited by **Type**. It retrieves a list of all attributes attached to an item. Here is one of its forms:

object[ ] GetCustomAttributes(bool *searchBases*)

If *searchBases* is true, then the attributes of all base classes through the inheritance chain will be included. Otherwise, only those classes defined by the specified type will be found.

The second method is **GetCustomAttribute( )**, which is defined by **Attribute**. One of its forms is shown here:

static Attribute GetCustomAttribute(MemberInfo *mi*, Type *attribtype*)

Here, *mi* is a **MemberInfo** object that describes the item for which the attributes are being obtained. The attribute desired is specified by *attribtype*. You will use this method when you know the name of the attribute you want to obtain, which is often the case. For example, assuming that the **UseAttrib** class has the **RemarkAttribute**, to obtain a reference to the **RemarkAttribute**, you can use a sequence like this:

```
// Get a MemberInfo instance associated with a
// class that has the RemarkAttribute.
Type t = typeof(UseAttrib);

// Retrieve the RemarkAttribute.
Type tRemAtt = typeof(RemarkAttribute);
RemarkAttribute ra = (RemarkAttribute)
      Attribute.GetCustomAttribute(t, tRemAtt);
```

This sequence works because **MemberInfo** is a base class of **Type**. Thus, **t** is a **MemberInfo** instance.

Once you have a reference to an attribute, you can access its members. This makes information associated with an attribute available to a program that uses an element to which an attribute is attached. For example, the following statement displays the **Remark** property:

```
Console.WriteLine(ra.Remark);
```

The following program puts together all of the pieces and demonstrates the use of **RemarkAttribute**:

```
// A simple attribute example.

using System;
using System.Reflection;

[AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute {
  string pri_remark; // underlies Remark property

  public RemarkAttribute(string comment) {
    pri_remark = comment;
  }

  public string Remark {
    get {
      return pri_remark;
    }
  }
}

[RemarkAttribute("This class uses an attribute.")]
class UseAttrib {
  // ...
}

class AttribDemo {
  static void Main() {
    Type t = typeof(UseAttrib);

    Console.Write("Attributes in " + t.Name + ": ");

    object[] attribs = t.GetCustomAttributes(false);
    foreach(object o in attribs) {
      Console.WriteLine(o);
    }

    Console.Write("Remark: ");

    // Retrieve the RemarkAttribute.
    Type tRemAtt = typeof(RemarkAttribute);
    RemarkAttribute ra = (RemarkAttribute)
          Attribute.GetCustomAttribute(t, tRemAtt);

    Console.WriteLine(ra.Remark);
  }
}
```

The output from the program is shown here:

```
Attributes in UseAttrib: RemarkAttribute
Remark: This class uses an attribute.
```

Positional vs. Named Parameters

In the preceding example, **RemarkAttribute** was initialized by passing the description string to the constructor, using the normal constructor syntax. In this case, the **comment** parameter to **RemarkAttribute( )** is called a *positional parameter.* This term relates to the fact that the argument is linked to a parameter by its position in the argument list. This is the way that all methods and constructors work in C#. For example, given a method called **test( )**, declared as shown here:

```
void test(int a, double b, string c)
```

the following call to **test( )**

```
test(10, 1.1, "hello");
```

passes 10 to **a**, 1.1 to **b**, and "hello" to **c** because of the position (i.e., order) of the arguments. However, for an attribute, you can also create *named parameters,* which can be assigned initial values by using their name. In this case, it is the name of the parameter, not its position, that is important.

A named parameter is supported by either a public field or property, which must be read-write and nonstatic. Any such field or property is automatically able to be used as a named parameter. A named parameter is given a value by an assignment statement that is located within the argument list when the attribute's constructor is invoked. Here is the general form of an attribute specification that includes named parameters:

[*attrib*(*positional-param-list*, *named-param1* = *value*, *named-param2* = *value*, ...)]

The positional parameters (if they exist) come first. Next, each named parameter is assigned a value. The order of the named parameters is not important. Named parameters do not need to be given a value. In this case, their default value will be used.

To understand how to use a named parameter, it is best to work through an example. Here is a version of **RemarkAttribute** that adds a field called **Supplement**, which can be used to hold a supplemental remark:

```
 [AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute {
  string pri_remark; // underlies Remark property

  // This can be used as a named parameter:
  public string Supplement;

  public RemarkAttribute(string comment) {
    pri_remark = comment;
    Supplement = "None";
  }

  public string Remark {
    get {
      return pri_remark;
    }
  }
}
```

As you can see, **Supplement** is initialized to the string "None" by the constructor. There is no way of using the constructor to assign it a different initial value. However, because **Supplement** is a public field of **RemarkAttribute**, it can be used as a named parameter, as shown here:

```
[RemarkAttribute("This class uses an attribute.",
        Supplement = "This is additional info.")]
class UseAttrib {
  // ...
}
```

Pay close attention to the way **RemarkAttribute**'s constructor is called. First, the positional argument is specified as it was before. Next is a comma, followed by the named parameter, **Supplement**, which is assigned a value. Finally, the closing **)** ends the call to the constructor. Thus, the named parameter is initialized within the call to the constructor. This syntax can be generalized. Position parameters must be specified in the order in which they appear. Named parameters are specified by assigning values to their name.

Here is a program that demonstrates the **Supplement** field:

```
// Use a named attribute parameter.

using System;
using System.Reflection;

[AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute {
  string pri_remark; // underlies Remark property

  public string Supplement; // this is a named parameter

  public RemarkAttribute(string comment) {
    pri_remark = comment;
    Supplement = "None";
  }

  public string Remark {
    get {
      return pri_remark;
    }
  }
}

[RemarkAttribute("This class uses an attribute.",
                 Supplement = "This is additional info.")]
class UseAttrib {
  // ...
}

class NamedParamDemo {
  static void Main() {
    Type t = typeof(UseAttrib);

    Console.Write("Attributes in " + t.Name + ": ");
```

```
    object[] attribs = t.GetCustomAttributes(false);
    foreach(object o in attribs) {
      Console.WriteLine(o);
    }

    // Retrieve the RemarkAttribute.
    Type tRemAtt = typeof(RemarkAttribute);
    RemarkAttribute ra = (RemarkAttribute)
          Attribute.GetCustomAttribute(t, tRemAtt);

    Console.Write("Remark: ");
    Console.WriteLine(ra.Remark);

    Console.Write("Supplement: ");
    Console.WriteLine(ra.Supplement);
  }
}
```

The output from the program is shown here:

```
Attributes in UseAttrib: RemarkAttribute
Remark: This class uses an attribute.
Supplement: This is additional info.
```

Before moving on, it is important to emphasize that **pri_remark** *cannot* be used as a named parameter because it is private to **RemarkAttribute**. The **Remark** property *cannot* be used as a named parameter because it is read-only. Remember that only public, read-write fields and properties can be used as named parameters.

A public, read-write property can be used as a named parameter in the same way as a field. For example, here an auto-implemented **int** property called **Priority** is added to **RemarkAttribute**:

```
// Use a property as a named attribute parameter.

using System;
using System.Reflection;

[AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute {
  string pri_remark; // underlies Remark property

  public string Supplement; // this is a named parameter

  public RemarkAttribute(string comment) {
    pri_remark = comment;
    Supplement = "None";
    Priority = 1;
  }

  public string Remark {
    get {
      return pri_remark;
    }
  }
```

```
  // Use a property as a named parameter.
  public int Priority { get; set; }
}

[RemarkAttribute("This class uses an attribute.",
                 Supplement = "This is additional info.",
                 Priority = 10)]
class UseAttrib {
  // ...
}

class NamedParamDemo {
  static void Main() {
    Type t = typeof(UseAttrib);

    Console.Write("Attributes in " + t.Name + ": ");

    object[] attribs = t.GetCustomAttributes(false);
    foreach(object o in attribs) {
      Console.WriteLine(o);
    }

    // Retrieve the RemarkAttribute.
    Type tRemAtt = typeof(RemarkAttribute);
    RemarkAttribute ra = (RemarkAttribute)
          Attribute.GetCustomAttribute(t, tRemAtt);

    Console.Write("Remark: ");
    Console.WriteLine(ra.Remark);

    Console.Write("Supplement: ");
    Console.WriteLine(ra.Supplement);

    Console.WriteLine("Priority: " + ra.Priority);
  }
}
```

The output is shown here:

```
Attributes in UseAttrib: RemarkAttribute
Remark: This class uses an attribute.
Supplement: This is additional info.
Priority: 10
```

There is one point of interest in the program. Notice the attribute specified before **UseAttrib** that is shown here:

```
[RemarkAttribute("This class uses an attribute.",
                 Supplement = "This is additional info.",
                 Priority = 10)]
```

The named attributes **Supplement** and **Priority** are *not* in any special order. These two assignments can be reversed without any change to the attribute.

One last point: For both positional and named parameters, the type of an attribute parameter must be either one of the built-in primitive types, **object**, **Type**, an enumeration, or a one-dimensional array of one of these types.

Three Built-in Attributes

C# defines several built-in attributes, but three are especially important because they apply to a wide variety of situations: **AttributeUsage**, **Conditional**, and **Obsolete**. They are examined here.

AttributeUsage

As mentioned earlier, the **AttributeUsage** attribute specifies the types of items to which an attribute can be applied. **AttributeUsage** is another name for the **System.AttributeUsageAttribute** class. **AttributeUsage** has the following constructor:

AttributeUsage(AttributeTargets *item*)

Here, *item* specifies the item or items upon which the attribute can be used. **AttributeTargets** is an enumeration that defines the following values:

All	Assembly	Class	Constructor
Delegate	Enum	Event	Field
GenericParameter	Interface	Method	Module
Parameter	Property	ReturnValue	Struct

Two or more of these value can be ORed together. For example, to specify an attribute that can be applied only to fields and properties, use

```
AttributeTargets.Field | AttributeTargets.Property
```

AttributeUsage supports two named parameters. The first is **AllowMultiple**, which is a **bool** value. If this value is true, then the attribute can be applied more than one time to a single item. The second is **Inherited**, which is also a **bool** value. If this value is true, then the attribute is inherited by derived classes. Otherwise, it is not inherited. The default setting is false for **AllowMultiple** and true for **Inherited**.

AttributeUsage also specifies a read-only property called **ValidOn**, which returns a value of type **AttributeTargets**, which specifies what types of items the attribute can be used on. The default is **AttributeTargets.All**.

The Conditional Attribute

The attribute **Conditional** is perhaps C#'s most interesting built-in attribute. It allows you to create *conditional methods.* A conditional method is invoked only when a specific symbol has been defined via **#define**. Otherwise, the method is bypassed. Thus, a conditional method offers an alternative to conditional compilation using **#if**.

Conditional is another name for **System.Diagnostics.ConditionalAttribute**. To use the **Conditional** attribute, you must include the **System.Diagnostics** namespace.

Let's begin with an example:

```
// Demonstrate the Conditional attribute.

#define TRIAL

using System;
using System.Diagnostics;

class Test {

  [Conditional("TRIAL")]
  void Trial() {
    Console.WriteLine("Trial version, not for distribution.");
  }

  [Conditional("RELEASE")]
  void Release() {
    Console.WriteLine("Final release version.");
  }

  static void Main() {
    Test t = new Test();

    t.Trial(); // called only if TRIAL is defined
    t.Release(); // called only if RELEASE is defined
  }
}
```

The output from this program is shown here:

```
Trial version, not for distribution.
```

Let's look closely at this program to understand why this output is produced. First, notice the program defines the symbol **TRIAL**. Next, notice how the methods **Trial( )** and **Release( )** are coded. They are both preceded with the **Conditional** attribute, which has this general form:

[Conditional *symbol*]

where *symbol* is the symbol that determines whether the method will be executed. This attribute can be used only on methods. If the symbol is defined, then when the method is called, it will be executed. If the symbol is not defined, then the method is not executed.

Inside **Main( )**, both **Trial( )** and **Release( )** are called. However, only **TRIAL** is defined. Thus, **Trial( )** is executed. The call to **Release( )** is ignored. If you define **RELEASE**, then **Release( )** will also be called. If you remove the definition for **TRIAL**, then **Trial( )** will not be called.

Conditional methods have a few restrictions. First, they must return **void**. Second, they must be members of a class or structure, not an interface. Third, they cannot be preceded with the **override** keyword.

The Obsolete Attribute

The **Obsolete** attribute, which is short for **System.ObsoleteAttribute**, lets you mark a program element as obsolete. It has this general form:

[Obsolete("*message*")]

Here, *message* is displayed when that program element is compiled. Here is a short example:

```
// Demonstrate the Obsolete attribute.

using System;

class Test {

  [Obsolete("Use MyMeth2, instead.")]
  public static int MyMeth(int a, int b) {
    return a / b;
  }

  // Improved version of MyMeth.
  public static int MyMeth2(int a, int b) {
    return b == 0 ? 0 : a /b;
  }

  static void Main() {
   // Warning displayed for this.
    Console.WriteLine("4 / 3 is " + Test.MyMeth(4, 3));

   // No warning here.
    Console.WriteLine("4 / 3 is " + Test.MyMeth2(4, 3));
  }
}
```

When the call to **MyMeth( )** is encountered in **Main( )** when this program is compiled, a warning will be generated that tells the user to use **MyMeth2( )** instead.

A second form of **Obsolete** is shown here:

[Obsolete("*message*", *error*)]

Here, *error* is a Boolean value. If it is true, then use of the obsolete item generates a compilation error rather than a warning. The difference is, of course, that a program containing an error cannot be compiled into an executable program.

This page intentionally left blank

18 CHAPTER

Generics

This chapter examines one of C#'s most sophisticated and powerful features: *generics.* Interestingly, although generics are now an indispensable part of C# programming, they were not included in the original 1.0 release. Instead, they were added by C# 2.0. It is not an overstatement to say that the addition of generics fundamentally changed the character of C#. Not only did it add a new syntactic element, it also added new capabilities and resulted in many changes and upgrades to the library. Although it has been a few years since the inclusion of generics in C#, the effects still reverberate throughout the language.

The generics feature is so important because it enables the creation of classes, structures, interfaces, methods, and delegates that work in a type-safe manner with various kinds of data. As you may know, many algorithms are logically the same no matter what type of data they are being applied to. For example, the mechanism that supports a queue is the same whether the queue is storing items of type **int**, **string**, **object**, or a user-defined class. Prior to generics, you might have created several different versions of the same algorithm to handle different types of data. Through the use of generics, you can define a solution once, independently of any specific type of data, and then apply that solution to a wide variety of data types without any additional effort.

This chapter describes the syntax, theory, and use of generics. It also shows how generics provide type safety for some previously difficult cases. Once you have completed this chapter, you will want to examine Chapter 24, which covers Collections. There you will find many examples of generics at work in the generic collection classes.

What Are Generics?

At its core, the term *generics* means *parameterized types.* Parameterized types are important because they enable you to create classes, structures, interfaces, methods, and delegates in which the type of data upon which they operate is specified as a parameter. Using generics, it is possible to create a single class, for example, that automatically works with different types of data. A class, structure, interface, method, or delegate that operates on a parameterized type is called *generic,* as in *generic class* or *generic method.*

It is important to understand that C# has always given you the ability to create generalized code by operating through references of type **object**. Because **object** is the base class of all other classes, an **object** reference can refer to any type of object. Thus, in pre-generics code, generalized code used **object** references to operate on a variety of different kinds of objects.

The problem was that it could not do so with type safety because casts were needed to convert between the **object** type and the actual type of the data. This was a potential source of errors because it was possible to accidentally use an incorrect cast. Generics avoid this problem by providing the type safety that was lacking. Generics also streamline the process because it is no longer necessary to employ casts to translate between **object** and the type of data that is actually being operated upon. Thus, generics expand your ability to re-use code, and let you do so safely and easily.

NOTE *A Warning to C++ and Java Programmers: Although C# generics are similar to templates in C++ and generics in Java, they are not the same as either. In fact, there are some fundamental differences among these three approaches to generics. If you have a background in C++ or Java, it is important to not jump to conclusions about how generics work in C#.*

A Simple Generics Example

Let's begin with a simple example of a generic class. The following program defines two classes. The first is the generic class **Gen**, and the second is **GenericsDemo**, which uses **Gen**.

```
// A simple generic class.

using System;

// In the following Gen class, T is a type parameter
// that will be replaced by a real type when an object
// of type Gen is created.
class Gen<T> {
  T ob; // declare a variable of type T

  // Notice that this constructor has a parameter of type T.
  public Gen(T o) {
    ob = o;
  }

  // Return ob, which is of type T.
  public T GetOb() {
    return ob;
  }

  // Show type of T.
  public void ShowType() {
    Console.WriteLine("Type of T is " + typeof(T));
  }
}

// Demonstrate the generic class.
class GenericsDemo {
  static void Main() {
    // Create a Gen reference for int.
    Gen<int> iOb;

    // Create a Gen<int> object and assign its reference to iOb.
    iOb = new Gen<int>(102);
```

```
    // Show the type of data used by iOb.
    iOb.ShowType();

    // Get the value in iOb.
    int v = iOb.GetOb();
    Console.WriteLine("value: " + v);

    Console.WriteLine();

    // Create a Gen object for strings.
    Gen<string> strOb = new Gen<string>("Generics add power.");

    // Show the type of data stored in strOb.
    strOb.ShowType();

    // Get the value in strOb.
    string str = strOb.GetOb();
    Console.WriteLine("value: " + str);
  }
}
```

The output produced by the program is shown here:

```
Type of T is System.Int32
value: 102

Type of T is System.String
value: Generics add power.
```

Let's examine this program carefully.

First, notice how **Gen** is declared by the following line.

```
class Gen<T> {
```

Here, **T** is the name of a *type parameter.* This name is used as a placeholder for the actual type that will be specified when a **Gen** object is created. Thus, **T** is used within **Gen** whenever the type parameter is needed. Notice that **T** is contained within **< >**. This syntax can be generalized. Whenever a type parameter is being declared, it is specified within angle brackets. Because **Gen** uses a type parameter, **Gen** is a *generic class.*

In the declaration of **Gen**, there is no special significance to the name **T**. Any valid identifier could have been used, but **T** is traditional. Other commonly used type parameter names include **V** and **E**. Of course, you can also use descriptive names for type parameters, such as **TValue** or **TKey**. When using a descriptive name, it is common practice to use **T** as the first letter.

Next, **T** is used to declare a variable called **ob**, as shown here:

```
T ob; // declare a variable of type T
```

As explained, **T** is a placeholder for the actual type that will be specified when a **Gen** object is created. Thus, **ob** will be a variable of the type *bound to* **T** when a **Gen** object is instantiated. For example, if type **string** is specified for **T**, then in that instance, **ob** will be of type **string**.

Now consider **Gen**'s constructor:

```
public Gen(T o) {
  ob = o;
}
```

Notice that its parameter, **o**, is of type **T**. This means that the actual type of **o** is determined by the type bound to **T** when a **Gen** object is created. Also, because both the parameter **o** and the instance variable **ob** are of type **T**, they will both be of the same actual type when a **Gen** object is created.

The type parameter **T** can also be used to specify the return type of a method, as is the case with the **GetOb( )** method, shown here:

```
public T GetOb() {
  return ob;
}
```

Because **ob** is also of type **T**, its type is compatible with the return type specified by **GetOb( )**.

The **ShowType( )** method displays the type of **T** by passing **T** to the **typeof** operator. Because a real type will be substituted for **T** when an object of type **Gen** is created, **typeof** will obtain type information about the actual type.

The **GenericsDemo** class demonstrates the generic **Gen** class. It first creates a version of **Gen** for type **int**, as shown here:

```
Gen<int> iOb;
```

Look closely at this declaration. First, notice that the type **int** is specified within the angle brackets after **Gen**. In this case, **int** is a *type argument* that is bound to **Gen**'s type parameter, **T**. This creates a version of **Gen** in which all uses of **T** are replaced by **int**. Thus, for this declaration, **ob** is of type **int**, and the return type of **GetOb( )** is of type **int**.

The next line assigns to **iOb** a reference to an instance of an **int** version of the **Gen** class:

```
iOb = new Gen<int>(102);
```

Notice that when the **Gen** constructor is called, the type argument **int** is also specified. This is necessary because the type of the variable (in this case **iOb**) to which the reference is being assigned is of type **Gen<int>**. Thus, the reference returned by **new** must also be of type **Gen<int>**. If it isn't, a compile-time error will result. For example, the following assignment will cause a compile-time error:

```
iOb = new Gen<double>(118.12); // Error!
```

Because **iOb** is of type **Gen<int>**, it can't be used to refer to an object of **Gen<double>**. This type checking is one of the main benefits of generics because it ensures type safety.

The program then displays the type of **ob** within **iOb**, which is **System.Int32**. This is the .NET structure that corresponds to **int**. Next, the program obtains the value of **ob** by use of the following line:

```
int v = iOb.GetOb();
```

Because the return type of **GetOb( )** is **T**, which was replaced by **int** when **iOb** was declared, the return type of **GetOb( )** is also **int**. Thus, this value can be assigned to an **int** variable.

Next, **GenericsDemo** declares an object of type **Gen<string>**:

```
Gen<string> strOb = new Gen<string>("Generics add power.");
```

Because the type argument is **string**, **string** is substituted for **T** inside **Gen**. This creates a **string** version of **Gen**, as the remaining lines in the program demonstrate.

Before moving on, a few terms need to be defined. When you specify a type argument such as **int** or **string** for **Gen**, you are creating what is referred to in C# as a *closed constructed type.* Thus, **Gen<int>** is a closed constructed type. In essence, a generic type, such as **Gen<T>**, is an abstraction. It is only after a specific version, such as **Gen<int>**, has been constructed that a concrete type has been created. In C# terminology, a construct such as **Gen<T>** is called an *open constructed type,* because the type parameter **T** (rather than an actual type, such as **int**) is specified.

More generally, C# defines the concepts of an *open type* and a *closed type.* An open type is a type parameter or any generic type whose type argument is (or involves) a type parameter. Any type that is not an open type is a closed type. A *constructed type* is a generic type for which all type arguments have been supplied. If those type arguments are all closed types, then it is a closed constructed type. If one or more of those type arguments are open types, it is an open constructed type.

Generic Types Differ Based on Their Type Arguments

A key point to understand about generic types is that a reference of one specific version of a generic type is not type-compatible with another version of the same generic type. For example, assuming the program just shown, the following line of code is in error and will not compile:

```
iOb = strOb; // Wrong!
```

Even though both **iOb** and **strOb** are of type **Gen<T>**, they are references to different types because their type arguments differ.

How Generics Improve Type Safety

At this point, you might be asking yourself the following question. Given that the same functionality found in the generic **Gen** class can be achieved without generics, by simply specifying **object** as the data type and employing the proper casts, what is the benefit of making **Gen** generic? The answer is that generics automatically ensure the type safety of all operations involving **Gen**. In the process, generics eliminate the need for you to use casts and type-check code by hand.

To understand the benefits of generics, first consider the following program that creates a non-generic equivalent of **Gen**:

```
// NonGen is functionally equivalent to Gen but does not use generics.

using System;

class NonGen {
  object ob; // ob is now of type object

  // Pass the constructor a reference of type object.
  public NonGen(object o) {
```

```
    ob = o;
  }

  // Return type object.
  public object GetOb() {
    return ob;
  }

  // Show type of ob.
  public void ShowType() {
    Console.WriteLine("Type of ob is " + ob.GetType());
  }
}

// Demonstrate the non-generic class.
class NonGenDemo {
  static void Main() {
    NonGen iOb;

    // Create NonGen object.
    iOb = new NonGen(102);

    // Show the type of data stored in iOb.
    iOb.ShowType();

    // Get the value in iOb.
    // This time, a cast is necessary.
    int v = (int) iOb.GetOb();
    Console.WriteLine("value: " + v);

    Console.WriteLine();

    // Create another NonGen object and store a string in it.
    NonGen strOb = new NonGen("Non-Generics Test");

    // Show the type of data stored in strOb.
    strOb.ShowType();

    // Get the value of strOb.
    // Again, notice that a cast is necessary.
    String str = (string) strOb.GetOb();
    Console.WriteLine("value: " + str);

    // This compiles, but is conceptually wrong!
    iOb = strOb;

    // The following line results in a runtime exception.
    // v = (int) iOb.GetOb(); // runtime error!
  }
}
```

This program produces the following output:

```
Type of ob is System.Int32
value: 102
```

```
Type of ob is System.String
value: Non-Generics Test
```

As you can see, the output is similar to the previous version of the program.

There are several things of interest in this version. First, notice that **NonGen** replaces all uses of **T** with **object**. This makes **NonGen** able to store any type of object, as can the generic version. However, this approach is bad for two reasons. First, explicit casts must be employed to retrieve the stored data. Second, many kinds of type mismatch errors cannot be found until runtime. Let's look closely at each problem.

We will begin with this line:

```
int v = (int) iOb.GetOb();
```

Because the return type of **GetOb( )** is now **object**, the cast to **int** is necessary to enable the value returned by **GetOb( )** to be unboxed and stored in **v**. If you remove the cast, the program will not compile. In the generic version of the program, this cast was not needed because **int** was specified as a type argument when **iOb** was constructed. In the non-generic version, the cast must be employed. This is not only an inconvenience, but a potential source of error.

Now, consider the following sequence from near the end of the program:

```
// This compiles, but is conceptually wrong!
iOb = strOb;

// The following line results in a runtime exception.
// v = (int) iOb.GetOb(); // runtime error!
```

Here, **strOb** is assigned to **iOb**. However, **strOb** refers to an object that contains a string, not an integer. This assignment is syntactically valid because all **NonGen** references are the same, and any **NonGen** reference can refer to any other **NonGen** object. However, the statement is semantically wrong, as the commented-out line shows. In that line, the return type of **GetOb( )** is cast to **int** and then an attempt is made to assign this value to **v**. The trouble is that **iOb** now refers to an object that stores a **string**, not an **int**. Unfortunately, without generics, the compiler won't catch this error. Instead, a runtime exception will occur when the cast to **int** is attempted. To see this for yourself, try removing the comment symbol from the start of the line and then compiling and running the program. A runtime error will occur.

The preceding sequence can't occur when generics are used. If this sequence were attempted in the generic version of the program, the compiler would catch it and report an error, thus preventing a serious bug that results in a runtime exception. The ability to create type-safe code in which type-mismatch errors are caught at compile time is a key advantage of generics. Although using **object** references to create "generic" code has always been possible in C#, that code was not type-safe and its misuse could result in runtime exceptions. Generics prevent this from occurring. In essence, through generics, what were once runtime errors have become compile-time errors. This is a major benefit.

There is one other point of interest in the **NonGen** program. Notice how the type of the **NonGen** instance variable **ob** is obtained by **ShowType( )**:

```
Console.WriteLine("Type of ob is " + ob.GetType());
```

Recall from Chapter 11 that **object** defines several methods that are available to all data types. One of these methods is **GetType( )**, which returns a **Type** object that describes the type of

the invoking object at runtime. Thus, even though the type of **ob** is specified as **object** in the program's source code, at runtime, the actual type of object being referred to is known. This is why the CLR will generate an exception if you try an invalid cast during program execution.

A Generic Class with Two Type Parameters

You can declare more than one type parameter in a generic type. To specify two or more type parameters, simply use a comma-separated list. For example, the following **TwoGen** class is a variation of the **Gen** class that has two type parameters:

```
// A simple generic class with two type parameters: T and V.

using System;

class TwoGen<T, V> {
  T ob1;
  V ob2;

  // Notice that this constructor has parameters of type T and V.
  public TwoGen(T o1, V o2) {
    ob1 = o1;
    ob2 = o2;
  }

  // Show types of T and V.
  public void showTypes() {
    Console.WriteLine("Type of T is " + typeof(T));
    Console.WriteLine("Type of V is " + typeof(V));
  }

  public T getob1() {
    return ob1;
  }

  public V GetObj2() {
    return ob2;
  }
}

// Demonstrate two generic type parameters.
class SimpGen {
  static void Main() {

    TwoGen<int, string> tgObj =
      new TwoGen<int, string>(119, "Alpha Beta Gamma");

    // Show the types.
    tgObj.showTypes();

    // Obtain and show values.
    int v = tgObj.getob1();
    Console.WriteLine("value: " + v);
```

```
    string str = tgObj.GetObj2();
    Console.WriteLine("value: " + str);
  }
}
```

The output from this program is shown here:

```
Type of T is System.Int32
Type of V is System.String
value: 119
value: Alpha Beta Gamma
```

Notice how **TwoGen** is declared:

```
class TwoGen<T, V> {
```

It specifies two type parameters: **T** and **V**, separated by a comma. Because it has two type parameters, two type arguments must be specified when a **TwoGen** object is created, as shown here:

```
TwoGen<int, string> tgObj =
  new TwoGen<int, string>(119, "Alpha Beta Gamma");
```

In this case, **int** is substituted for **T** and **string** is substituted for **V**.

Although the two type arguments differ in this example, it is possible for both types to be the same. For example, the following line of code is valid:

```
TwoGen<string, string> x = new TwoGen<string, string>("Hello", "Goodbye");
```

In this case, both **T** and **V** would be of type **string**. Of course, if the type arguments were always the same, then two type parameters would be unnecessary.

The General Form of a Generic Class

The generics syntax shown in the preceding examples can be generalized. Here is the syntax for declaring a generic class:

class *class-name*<*type-param-list*> { // ...

Here is the syntax for declaring a reference to a generics class:

class-name<*type-arg-list*> *var-name* =
    new *class-name*<*type-arg-list*>(*cons-arg-list*);

Constrained Types

In the preceding examples, the type parameters could be replaced by any type. For example, given this declaration

```
class Gen<T> {
```

any type can be specified for **T**. Thus, it is legal to create **Gen** objects in which **T** is replaced by **int**, **double**, **string**, **FileStream**, or any other type. Although having no restrictions on the

type argument is fine for many purposes, sometimes it is useful to limit the types that can be used as a type argument. For example, you might want to create a method that operates on the contents of a stream, including a **FileStream** or **MemoryStream**. This situation seems perfect for generics, but you need some way to ensure that only stream types are used as type arguments. You don't want to allow a type argument of **int**, for example. You also need some way to tell the compiler that the methods defined by a stream will be available for use. For example, your generic code needs some way to know that it can call the **Read( )** method.

To handle such situations, C# provides *constrained types.* When specifying a type parameter, you can specify a constraint that the type parameter must satisfy. This is accomplished through the use of a **where** clause when specifying the type parameter, as shown here:

class *class-name*<*type-param*> where *type-param* : *constraints* { // ...

Here, *constraints* is a comma-separated list of constraints.

C# defines the following types of constraints.

- You can require that a certain base class be present in a type argument by using a *base class constraint.* This constraint is specified by naming the desired base class. There is a variation of this constraint, called a *naked type constraint,* in which a type parameter (rather than an actual type) specifies the base class. This enables you to establish a relationship between two type parameters.
- You can require that one or more interfaces be implemented by a type argument by using an *interface constraint.* This constraint is specified by naming the desired interface.
- You can require that the type argument supply a parameterless constructor. This is called a *constructor constraint.* It is specified by **new( )**.
- You can specify that a type argument must be a reference type by specifying the *reference type constraint:* **class**.
- You can specify that the type argument be a value type by specifying the *value type constraint:* **struct**.

Of these constraints, the base class constraint and the interface constraint are probably the most often used, but all are important. Each constraint is examined in the following sections.

Using a Base Class Constraint

The base class constraint enables you to specify a base class that a type argument must inherit. A base class constraint serves two important purposes. First, it lets you use the members of the base class specified by the constraint within the generic class. For example, you can call a method or use a property of the base class. Without a base class constraint, the compiler has no way to know what type of members a type argument might have. By supplying a base class constraint, you are letting the compiler know that all type arguments will have the members defined by that base class.

The second purpose of a base class constraint is to ensure that only type arguments that support the specified base class are used. This means that for any given base class constraint, the type argument must be either the base class, itself, or a class derived from that base class. If you attempt to use a type argument that does not match or inherit the specified base class, a compile-time error will result.

The base class constraint uses this form of the **where** clause:

where *T* : *base-class-name*

Here, *T* is the name of the type parameter, and *base-class-name* is the name of the base class. Only one base class can be specified.

Here is a simple example that demonstrates the base class constraint mechanism:

```
// A simple demonstration of a base class constraint.

using System;

class A {
  public void Hello() {
    Console.WriteLine("Hello");
  }
}

// Class B inherits A.
class B : A { }

// Class C does not inherit A.
class C { }

// Because of the base class constraint, all type arguments
// passed to Test must have A as a base class.
class Test<T> where T : A {
  T obj;

  public Test(T o) {
    obj = o;
  }

  public void SayHello() {
    // OK to call Hello() because it's declared
    // by the base class A.
    obj.Hello();
  }
}

class BaseClassConstraintDemo {
  static void Main() {
    A a = new A();
    B b = new B();
    C c = new C();

    // The following is valid because A is the specified base class.
    Test<A> t1 = new Test<A>(a);

    t1.SayHello();

    // The following is valid because B inherits A.
    Test<B> t2 = new Test<B>(b);
```

```
    t2.SayHello();

    // The following is invalid because C does not inherit A.
//    Test<C> t3 = new Test<C>(c); // Error!
//    t3.SayHello(); // Error!
  }
}
```

In this program, class **A** is inherited by **B**, but not by **C**. Notice also that **A** declares a method called **Hello( )**. Next, notice that **Test** is a generic class that is declared like this:

```
class Test<T> where T : A {
```

The **where** clause stipulates that any type argument specified for **T** must have **A** as a base class.

Now notice that **Test** declares the method **SayHello( )**, shown next:

```
public void SayHello() {
  // OK to call Hello() because it's declared
  // by the base class A.
  obj.Hello();
}
```

This method calls **Hello( )** on **obj**, which is a **T** object. The key point is that the only reason that **Hello( )** can be called is because the base class constraint requires that any type argument bound to **T** must be **A** or inherit **A**, and **A** declares **Hello( )**. Thus, any valid **T** will define **Hello( )**. If the base class constraint had not been used, the compiler would have no way of knowing that a method called **Hello( )** could be called on a object of type **T**. You can prove this for yourself by removing the **where** clause. The program will no longer compile because the **Hello( )** method will be unknown.

In addition to enabling access to members of the base class, the base class constraint enforces that only types that inherit the base class can be passed as type arguments. This is why the following two lines are commented-out:

```
//    Test<C> t3 = new Test<C>(c); // Error!
//    t3.SayHello(); // Error!
```

Because **C** does not inherit **A**, it can't be used as a type argument when constructing a **Test** object. You can prove this by removing the comment symbols and trying to recompile.

Before continuing, let's review the two effects of a base class constraint: A base class constraint enables a generic class to access the members of the base class. It also ensures that only those type arguments that fulfill this constraint are valid, thus preserving type safety.

Although the preceding example shows the "how" of base class constraints, it does not show the "why." To better understand the value of base type constraints, let's work through another, more practical example. Assume you want to create a mechanism that manages lists of telephone numbers. Furthermore, assume you want to use different lists for different groupings of numbers. Specifically, you want one list for friends, another for suppliers, and so on. To accomplish this, you might start by creating a base class called **PhoneNumber** that stores a name and a phone number linked to that name. Such a class might look like this:

```
// A base class that stores a name and phone number.
class PhoneNumber {
  public PhoneNumber(string n, string num) {
    Name = n;
    Number = num;
  }

  // Auto-implemented properties that hold a name and phone number.
  public string Number { get; set; }
  public string Name { get; set; }
}
```

Next, you create two classes that inherit **PhoneNumber**: **Friend** and **Supplier**. They are shown here:

```
// A class of phone numbers for friends.
class Friend : PhoneNumber {

  public Friend(string n, string num, bool wk) :
    base(n, num)
  {
    IsWorkNumber = wk;
  }

  public bool IsWorkNumber { get; private set; }
  // ...
}

// A class of phone numbers for suppliers.
class Supplier : PhoneNumber {
  public Supplier(string n, string num) :
    base(n, num) { }

  // ...
}
```

Notice that **Friend** adds a property called **IsWorkNumber**, which returns true if the telephone number is a work number.

To manage telephone lists, you create a class called **PhoneList**. Because you want this class to manage any type of phone list, you make it generic. Furthermore, because part of the list management is looking up numbers given names, and vice versa, you add the constraint that requires that the type of objects stored in the list must be instances of a class derived from **PhoneNumber**.

```
// PhoneList can manage any type of phone list
// as long as it is derived from PhoneNumber.
class PhoneList<T> where T : PhoneNumber {
  T[] phList;
  int end;

  public PhoneList() {
    phList = new T[10];
```

```
    end = 0;
  }

  // Add an entry to the list.
  public bool Add(T newEntry) {
    if(end == 10) return false;

    phList[end] = newEntry;
    end++;

    return true;
  }

  // Given a name, find and return the phone info.
  public T FindByName(string name) {

    for(int i=0; i<end; i++) {
      // Name can be used because it is a member of
      // PhoneNumber, which is the base class constraint.
      if(phList[i].Name == name)
        return phList[i];
    }

    // Name not in list.
    throw new NotFoundException();
  }

  // Given a number, find and return the phone info.
  public T FindByNumber(string number) {

    for(int i=0; i<end; i++) {
      // Number can be used because it is also a member of
      // PhoneNumber, which is the base class constraint.
      if(phList[i].Number == number)
        return phList[i];
    }

    // Number not in list.
    throw new NotFoundException();
  }

  // ...
}
```

The base class constraint enables code inside **PhoneList** to access the properties **Name** and **Number** for any type of telephone list. It also guarantees that only valid types are used to construct a **PhoneList** object. Notice that **PhoneList** throws a **NotFoundException** if a name or number is not found. This is a custom exception that is declared as shown here:

```
class NotFoundException : Exception {
  /* Implement all of the Exception constructors. Notice that
     the constructors simply execute the base class constructor.
     Because NotFoundException adds nothing to Exception,
     there is no need for any further actions. */
```

```
  public NotFoundException() : base() { }
  public NotFoundException(string str) : base(str) { }
  public NotFoundException(string str, Exception inner) :
    base(str, inner) { }
  protected NotFoundException(
    System.Runtime.Serialization.SerializationInfo si,
    System.Runtime.Serialization.StreamingContext sc) :
      base(si, sc) { }
}
```

Although only the default constructor is used by this example, **NotFoundException** implements all of the constructors defined by **Exception** for the sake of illustration. Notice that these constructors simply invoke the equivalent base class constructor defined by **Exception**. Because **NotFoundException** adds nothing to **Exception**, there is no reason for any further action.

The following program puts together all the pieces and demonstrates **PhoneList**. Notice that a class called **EmailFriend** is also created. This class does not inherit **PhoneNumber**. Thus, it *cannot* be used to create a **PhoneList**.

```
// A more practical demonstration of a base class constraint.

using System;

// A custom exception that is thrown if a name or number is not found.
class NotFoundException : Exception {
  /* Implement all of the Exception constructors. Notice that
     the constructors simply execute the base class constructor.
     Because NotFoundException adds nothing to Exception,
     there is no need for any further actions. */
  public NotFoundException() : base() { }
  public NotFoundException(string str) : base(str) { }
  public NotFoundException(string str, Exception inner) :
    base(str, inner) { }
  protected NotFoundException(
    System.Runtime.Serialization.SerializationInfo si,
    System.Runtime.Serialization.StreamingContext sc) :
      base(si, sc) { }
}
// A base class that stores a name and phone number.
class PhoneNumber {

  public PhoneNumber(string n, string num) {
    Name = n;
    Number = num;
  }

  public string Number { get; set; }
  public string Name { get; set; }

}

// A class of phone numbers for friends.
class Friend : PhoneNumber {
```

```
  public Friend(string n, string num, bool wk) :
    base(n, num)
  {
    IsWorkNumber = wk;
  }

  public bool IsWorkNumber { get; private set; }

  // ...
}

// A class of phone numbers for suppliers.
class Supplier : PhoneNumber {
  public Supplier(string n, string num) :
    base(n, num) { }

  // ...
}

// Notice that this class does not inherit PhoneNumber.
class EmailFriend {
  // ...
}

// PhoneList can manage any type of phone list
// as long as it is derived from PhoneNumber.
class PhoneList<T> where T : PhoneNumber {
  T[] phList;
  int end;

  public PhoneList() {
    phList = new T[10];
    end = 0;
  }

  // Add an entry to the list.
  public bool Add(T newEntry) {
    if(end == 10) return false;

    phList[end] = newEntry;
    end++;

    return true;
  }

  // Given a name, find and return the phone info.
  public T FindByName(string name) {

    for(int i=0; i<end; i++) {
      // Name can be used because it is a member of
      // PhoneNumber, which is the base class constraint.
      if(phList[i].Name == name)
        return phList[i];
    }
```

```
    // Name not in list.
    throw new NotFoundException();
  }

  // Given a number, find and return the phone info.
  public T FindByNumber(string number) {

    for(int i=0; i<end; i++) {
      // Number can be used because it is also a member of
      // PhoneNumber, which is the base class constraint.
      if(phList[i].Number == number)
        return phList[i];
    }

    // Number not in list.
    throw new NotFoundException();
  }

  // ...
}

// Demonstrate base class constraints.
class UseBaseClassConstraint {
  static void Main() {

    // The following code is OK because Friend
    // inherits PhoneNumber.
    PhoneList<Friend> plist = new PhoneList<Friend>();
    plist.Add(new Friend("Tom", "555-1234", true));
    plist.Add(new Friend("Gary", "555-6756", true));
    plist.Add(new Friend("Matt", "555-9254", false));

    try {
      // Find the number of a friend given a name.
      Friend frnd = plist.FindByName("Gary");

      Console.Write(frnd.Name + ": " + frnd.Number);

      if(frnd.IsWorkNumber)
        Console.WriteLine(" (work)");
      else
        Console.WriteLine();
    } catch(NotFoundException) {
      Console.WriteLine("Not Found");
    }

    Console.WriteLine();

    // The following code is also OK because Supplier
    // inherits PhoneNumber.
    PhoneList<Supplier> plist2 = new PhoneList<Supplier>();
    plist2.Add(new Supplier("Global Hardware", "555-8834"));
    plist2.Add(new Supplier("Computer Warehouse", "555-9256"));
    plist2.Add(new Supplier("NetworkCity", "555-2564"));
```

```
    try {
      // Find the name of a supplier given a number.
      Supplier sp = plist2.FindByNumber("555-2564");
      Console.WriteLine(sp.Name + ": " + sp.Number);
    } catch(NotFoundException) {
        Console.WriteLine("Not Found");
    }

    // The following declaration is invalid because EmailFriend
    // does NOT inherit PhoneNumber.
//    PhoneList<EmailFriend> plist3 =
//        new PhoneList<EmailFriend>(); // Error!
  }
}
```

The output from the program is shown here:

```
Gary: 555-6756 (work)

NetworkCity: 555-2564
```

You might want to try experimenting with this program a bit. For example, try creating different types of telephone lists. Also, try using **IsWorkNumber** from within **PhoneList**. As you will see, the compiler won't let you do it. The reason is that **IsWorkNumber** is a property defined by **Friend**, not by **PhoneNumber**. Thus, **PhoneList** has no knowledge of it.

Using an Interface Constraint

The interface constraint enables you to specify an interface that a type argument must implement. The interface constraint serves the same two important purposes as the base class constraint. First, it lets you use the members of the interface within the generic class. Second, it ensures that only type arguments that implement the specified interface are used. This means that for any given interface constraint, the type argument must be either the interface or a type that implements that interface.

The interface constraint uses this form of the **where** clause:

where *T* : *interface-name*

Here, *T* is the name of the type parameter, and *interface-name* is the name of the interface. More than one interface can be specified by using a comma-separated list. If a constraint includes both a base class and interface, then the base class must be listed first.

The following program illustrates the interface constraint by reworking the telephone list example shown in the previous section. In this version, the **PhoneNumber** class has been converted into an interface called **IPhoneNumber**. This interface is then implemented by **Friend** and **Supplier**.

```
// Use an interface constraint.

using System;

// A custom exception that is thrown if a name or number is not found.
class NotFoundException : Exception {
  /* Implement all of the Exception constructors. Notice that
```

```
     the constructors simply execute the base class constructor.
     Because NotFoundException adds nothing to Exception,
     there is no need for any further actions. */
  public NotFoundException() : base() { }
  public NotFoundException(string str) : base(str) { }
  public NotFoundException(string str, Exception inner) :
    base(str, inner) { }
  protected NotFoundException(
    System.Runtime.Serialization.SerializationInfo si,
    System.Runtime.Serialization.StreamingContext sc) :
      base(si, sc) { }
}

// An interface that supports a name and phone number.
public interface IPhoneNumber {

  string Number {
    get;
    set;
  }

  string Name {
    get;
    set;
  }
}

// A class of phone numbers for friends.
// It implements IPhoneNumber.
class Friend : IPhoneNumber {

  public Friend(string n, string num, bool wk) {
    Name = n;
    Number = num;

    IsWorkNumber = wk;
  }

  public bool IsWorkNumber { get; private set; }

  // Implement IPhoneNumber.
  public string Number { get; set; }
  public string Name { get; set; }

  // ...
}

// A class of phone numbers for suppliers.
class Supplier : IPhoneNumber {

  public Supplier(string n, string num) {
    Name = n;
    Number = num;
  }
```

```
  // Implement IPhoneNumber.
  public string Number { get; set; }
  public string Name { get; set; }

  // ...
}

// Notice that this class does not implement IPhoneNumber.
class EmailFriend {
  // ...
}

// PhoneList can manage any type of phone list
// as long as it implements IPhoneNumber.
class PhoneList<T> where T : IPhoneNumber {
  T[] phList;
  int end;

  public PhoneList() {
    phList = new T[10];
    end = 0;
  }

  public bool Add(T newEntry) {
    if(end == 10) return false;

    phList[end] = newEntry;
    end++;

    return true;
  }

  // Given a name, find and return the phone info.
  public T FindByName(string name) {

    for(int i=0; i<end; i++) {
      // Name can be used because it is a member of
      // IPhoneNumber, which is the interface constraint.
      if(phList[i].Name == name)
        return phList[i];
    }

    // Name not in list.
    throw new NotFoundException();
  }

  // Given a number, find and return the phone info.
  public T FindByNumber(string number) {

    for(int i=0; i<end; i++) {
      // Number can be used because it is also a member of
      // IPhoneNumber, which is the interface constraint.
      if(phList[i].Number == number)
        return phList[i];
    }
```

```
    // Number not in list.
    throw new NotFoundException();
  }

  // ...
}

// Demonstrate interface constraints.
class UseInterfaceConstraint {
  static void Main() {

    // The following code is OK because Friend
    // implements IPhoneNumber.
    PhoneList<Friend> plist = new PhoneList<Friend>();
    plist.Add(new Friend("Tom", "555-1234", true));
    plist.Add(new Friend("Gary", "555-6756", true));
    plist.Add(new Friend("Matt", "555-9254", false));

    try {
      // Find the number of a friend given a name.
      Friend frnd = plist.FindByName("Gary");

      Console.Write(frnd.Name + ": " + frnd.Number);

      if(frnd.IsWorkNumber)
        Console.WriteLine(" (work)");
      else
        Console.WriteLine();
    } catch(NotFoundException) {
      Console.WriteLine("Not Found");
    }

    Console.WriteLine();

    // The following code is also OK because Supplier
    // implements IPhoneNumber.
    PhoneList<Supplier> plist2 = new PhoneList<Supplier>();
    plist2.Add(new Supplier("Global Hardware", "555-8834"));
    plist2.Add(new Supplier("Computer Warehouse", "555-9256"));
    plist2.Add(new Supplier("NetworkCity", "555-2564"));

    try {
      // Find the name of a supplier given a number.
      Supplier sp = plist2.FindByNumber("555-2564");
      Console.WriteLine(sp.Name + ": " + sp.Number);
    } catch(NotFoundException) {
        Console.WriteLine("Not Found");
    }

    // The following declaration is invalid because EmailFriend
    // does NOT implement IPhoneNumber.
//    PhoneList<EmailFriend> plist3 =
//        new PhoneList<EmailFriend>(); // Error!
  }
}
```

In this version of the program, the interface constraint specified by **PhoneList** requires that a type argument implement the **IPhoneList** interface. Because both **Friend** and **Supplier** implement **IPhoneList**, they are valid types to be bound to **T**. However, **EmailFriend** does not implement **IPhoneList** and cannot be bound to **T**. To prove this, remove the comment symbols from the last two lines in **Main( )**. As you will see, the program will not compile.

Using the new( ) Constructor Constraint

The **new( )** constructor constraint enables you to instantiate an object of a generic type. Normally, you cannot create an instance of a generic type parameter. However, the **new( )** constraint changes this because it requires that a type argument supply a parameterless constructor. This can be the default constructor provided automatically when no explicit constructor is declared or a parameterless constructor explicitly defined by you. With the **new( )** constraint in place, you can invoke the parameterless constructor to create an object.

Here is a simple example that illustrates the use of **new( )**:

```
// Demonstrate a new() constructor constraint.

using System;

class MyClass {

  public MyClass() {
    // ...
  }

  //...
}

class Test<T> where T : new() {
  T obj;

  public Test() {
    // This works because of the new() constraint.
    obj = new T(); // create a T object
  }

  // ...
}

class ConsConstraintDemo {
  static void Main() {

    Test<MyClass> x = new Test<MyClass>();

  }
}
```

First, notice the declaration of the **Test** class, shown here:

```
class Test<T> where T : new() {
```

Because of the **new( )** constraint, any type argument must supply a parameterless constructor.

Next, examine the **Test** constructor, shown here:

```
public Test() {
  // This works because of the new() constraint.
  obj = new T(); // create a T object
}
```

A new object of type **T** is created and a reference to it is assigned to **obj**. This statement is valid only because the **new( )** constraint ensures that a constructor will be available. To prove this, try removing the **new( )** constraint and then attempt to recompile the program. As you will see, an error will be reported.

In **Main( )**, an object of type **Test** is instantiated, as shown here:

```
Test<MyClass> x = new Test<MyClass>();
```

Notice that the type argument is **MyClass**, and that **MyClass** defines a parameterless constructor. Thus, it is valid for use as a type argument for **Test**. It must be emphasized that it was not necessary for **MyClass** to explicitly declare a parameterless constructor. Its default constructor would also satisfy the constraint. However, if a class needs other constructors in addition to a parameterless one, then it would be necessary to also explicitly declare a parameterless version, too.

There are three important points about using **new( )**. First, it can be used with other constraints, but it must be the last constraint in the list. Second, **new( )** allows you to construct an object using only the parameterless constructor, even when other constructors are available. In other words, it is not permissible to pass arguments to the constructor of a type parameter. Third, you cannot use **new( )** in conjunction with a value type constraint, described next.

The Reference Type and Value Type Constraints

The next two constraints enable you to indicate that a type argument must be either a reference type or a value type. These are useful in the few cases in which the difference between reference and value types are important to generic code. Here is the general form of the reference type constraint:

where *T* : class

In this form of the **where** clause, the keyword **class** specifies that *T* must be a reference type. Thus, an attempt to use a value type, such as **int** or **bool**, for *T* will result in a compilation error.

Here is the general form of the value type constraint:

where *T* : struct

In this case, the keyword **struct** specifies that *T* must be a value type. (Recall that structures are value types.) Thus, an attempt to use a reference type, such as **string**, for *T* will result in a compilation error. In both cases, when additional constraints are present, **class** or **struct** must be the first constraint in the list.

Here is an example that demonstrates the reference type constraint:

```
// Demonstrate a reference constraint.

using System;

class MyClass {
  //...
}

// Use a reference constraint.
class Test<T> where T : class {
  T obj;

  public Test() {
    // The following statement is legal only
    // because T is guaranteed to be a reference
    // type, which can be assigned the value null.
    obj = null;
  }

  // ...
}

class ClassConstraintDemo {
  static void Main() {

    // The following is OK because MyClass is a class.
    Test<MyClass> x = new Test<MyClass>();

    // The next line is in error because int is a value type.
//    Test<int> y = new Test<int>();
  }
}
```

First, notice how **Test** is declared:

```
class Test<T> where T : class {
```

The **class** constraint requires that any type argument for **T** be a reference type. In this program, this is necessary because of what occurs inside the **Test** constructor:

```
public Test() {
  // The following statement is legal only
  // because T is guaranteed to be a reference
  // type, which can be assigned the value null.
  obj = null;
}
```

Here, **obj** (which is of type **T**) is assigned the value **null**. This assignment is valid only for reference types. As a general rule, you cannot assign **null** to a value type. (The exception to this rule is the *nullable type*, which is a special structure type that encapsulates a value type and allows the value **null**. See Chapter 20 for details.) Therefore, without the constraint, the

assignment would not have been valid and the compile would have failed. This is one case in which the difference between value types and reference types might be important to a generic routine.

The value type constraint is the complement of the reference type constraint. It simply ensures that any type argument is a value type, including a **struct** or an **enum**. (In this context, a nullable type is not considered a value type.) Here is an example:

```
// Demonstrate a value type constraint.

using System;

struct MyStruct {
  //...
}

class MyClass {
  // ...
}

class Test<T> where T : struct {
  T obj;

  public Test(T x) {
    obj = x;
  }

  // ...
}

class ValueConstraintDemo {
  static void Main() {

    // Both of these declarations are legal.

    Test<MyStruct> x = new Test<MyStruct>(new MyStruct());

    Test<int> y = new Test<int>(10);

    // But, the following declaration is illegal!
//    Test<MyClass> z = new Test<MyClass>(new MyClass());
  }
}
```

In this program, **Test** is declared as shown here:

```
class Test<T> where T : struct {
```

Because **T** of **Test** now has the **struct** constraint, **T** can be bound only to value type arguments. This means that **Test<MyStruct>** and **Test<int>** are valid, but **Test<MyClass>** is not. To prove this, try removing the comment symbols from the start of the last line in the program and recompiling. An error will be reported.

Using a Constraint to Establish a Relationship Between Two Type Parameters

There is a variation of the base class constraint that allows you to establish a relationship between two type parameters. For example, consider the following generic class declaration:

```
class Gen<T, V> where V : T {
```

In this declaration, the **where** clause tells the compiler that the type argument bound to **V** must be identical to or inherit from the type argument bound to **T**. If this relationship is not present when an object of type **Gen** is declared, then a compile-time error will result. A constraint that uses a type parameter, such as that just shown, is called a *naked type constraint*. The following example illustrates this constraint:

```
// Create relationship between two type parameters.

using System;

class A {
  //...
}

class B : A {
  // ...
}

// Here, V must be or inherit from T.
class Gen<T, V> where V : T {
  // ...
}

class NakedConstraintDemo {
  static void Main() {

    // This declaration is OK because B inherits A.
    Gen<A, B> x = new Gen<A, B>();

    // This declaration is in error because
    // A does not inherit B.
//    Gen<B, A> y = new Gen<B, A>();

  }
}
```

First, notice that class **B** inherits class **A**. Next, examine the two **Gen** declarations in **Main( )**. As the comments explain, the first declaration

```
Gen<A, B> x = new Gen<A, B>();
```

is legal because **B** inherits **A**. However, the second declaration

```
//    Gen<B, A> y = new Gen<B, A>();
```

is illegal because **A** does not inherit **B**.

Using Multiple Constraints

There can be more than one constraint associated with a type parameter. When this is the case, use a comma-separated list of constraints. In this list, the first constraint must be **class** or **struct** (if present) or the base class (if one is specified). It is illegal to specify both a **class** or **struct** constraint and a base class constraint. Next in the list must be any interface constraints. The **new( )** constraint must be last. For example, this is a valid declaration.

```
class Gen<T> where T : MyClass, IMyInterface, new() { // ...
```

In this case, **T** must be replaced by a type argument that inherits **MyClass**, implements **IMyInterface**, and has a parameterless constructor.

When using two or more type parameters, you can specify a constraint for each parameter by using a separate **where** clause. Here is an example:

```
// Use multiple where clauses.

using System;

// Gen has two type arguments and both have a where clause.
class Gen<T, V> where T : class
                where V : struct {
  T ob1;
  V ob2;

  public Gen(T t, V v) {
    ob1 = t;
    ob2 = v;
  }
}

class MultipleConstraintDemo {
  static void Main() {
    // This is OK because string is a class and
    // int is a value type.
    Gen<string, int> obj = new Gen<string, int>("test", 11);

    // The next line is wrong because bool is not
    // a reference type.
//    Gen<bool, int> obj = new Gen<bool, int>(true, 11);
  }
}
```

In this example, **Gen** takes two type arguments and both have a **where** clause. Pay special attention to its declaration:

```
class Gen<T, V> where T : class
                where V : struct {
```

Notice the only thing that separates the first **where** clause from the second is whitespace. No other punctuation is required or valid.

Creating a Default Value of a Type Parameter

When writing generic code, there will be times when the difference between value types and reference types is an issue. One such situation occurs when you want to give a variable of a type parameter a default value. For reference types, the default value is **null**. For non-**struct** value types, the default value is 0. The default value for a **struct** is an object of that **struct** with all fields set to their defaults. Thus, trouble occurs if you want to give a variable of a type parameter a default value. What value would you use: **null**, 0, or something else?

For example, given a generic class called **Test** declared like this:

```
class Test<T> {
  T obj;
  // ...
```

if you want to give **obj** a default value, would you use

```
obj = null; // works only for reference types
```

or

```
obj = 0; // works only for numeric types and enums, but not structs
```

The solution to this problem is to use another form of **default**, shown here:

default(*type*)

This is the operator form of **default**, and it produces a default value of the specified *type*, no matter what type is used. Thus, continuing with the example, to assign **obj** a default value of type **T**, you would use this statement:

```
obj = default(T);
```

This will work for all type arguments, whether they are value or reference types.

Here is a short program that demonstrates **default**:

```
// Demonstrate the default operator.

using System;

class MyClass {
  //...
}

// Construct a default value of T.
class Test<T> {
  public T obj;

  public Test() {
    // The following statement would work only for reference types.
//    obj = null; // can't use

    // The following statement will work only for numeric value types.
//    obj = 0; // can't use
```

```
    // This statement works for both reference and value types.
    obj = default(T); // Works!
  }

  // ...
}

class DefaultDemo {
  static void Main() {

    // Construct Test using a reference type.
    Test<MyClass> x = new Test<MyClass>();

    if(x.obj == null)
      Console.WriteLine("x.obj is null.");

    // Construct Test using a value type.
    Test<int> y = new Test<int>();

    if(y.obj == 0)
      Console.WriteLine("y.obj is 0.");
  }
}
```

The output is shown here:

```
x.obj is null.
y.obj is 0.
```

Generic Structures

C# allows you to create generic structures. The syntax is the same as for generic classes. For example, in the following program, the **XY** structure, which stores X, Y coordinates, is generic:

```
// Demonstrate a generic struct.
using System;

// This structure is generic.
struct XY<T> {
  T x;
  T y;

  public XY(T a, T b) {
    x = a;
    y = b;
  }

  public T X {
    get { return x; }
    set { x = value; }
  }

  public T Y {
    get { return y; }
```

```
    set { y = value; }
  }
}

class StructTest {
  static void Main() {
    XY<int> xy = new XY<int>(10, 20);
    XY<double> xy2 = new XY<double>(88.0, 99.0);

    Console.WriteLine(xy.X + ", " + xy.Y);

    Console.WriteLine(xy2.X + ", " + xy2.Y);
  }
}
```

The output is shown here:

```
10, 20
88, 99
```

Like generic classes, generic structures can have constraints. For example, this version of **XY** restricts type arguments to value types:

```
struct XY<T> where T : struct {
// ...
```

Creating a Generic Method

As the preceding examples have shown, methods inside a generic class can make use of a class' type parameter and are, therefore, automatically generic relative to the type parameter. However, it is possible to declare a generic method that uses one or more type parameters of its own. Furthermore, it is possible to create a generic method that is enclosed within a non-generic class.

Let's begin with an example. The following program declares a non-generic class called **ArrayUtils** and a static generic method within that class called **CopyInsert( )**. The **CopyInsert( )** method copies the contents of one array to another, inserting a new element at a specified location in the process. It can be used with any type of array.

```
// Demonstrate a generic method.

using System;

// A class of array utilities. Notice that this is not
// a generic class.
class ArrayUtils {

  // Copy an array, inserting a new element
  // in the process. This is a generic method.
  public static bool CopyInsert<T>(T e, uint idx,
                                   T[] src, T[] target) {

    // See if target array is big enough.
    if(target.Length < src.Length+1)
      return false;
```

```
    // Copy src to target, inserting e at idx in the process.
    for(int i=0, j=0; i < src.Length; i++, j++) {
      if(i == idx) {
        target[j] = e;
        j++;
      }
      target[j] = src[i];
    }

    return true;
  }
}

class GenMethDemo {
  static void Main() {
    int[] nums = { 1, 2, 3 };
    int[] nums2 = new int[4];

    // Display contents of nums.
    Console.Write("Contents of nums: ");
    foreach(int x in nums)
      Console.Write(x + " ");

    Console.WriteLine();

    // Operate on an int array.
    ArrayUtils.CopyInsert(99, 2, nums, nums2);

    // Display contents of nums2.
    Console.Write("Contents of nums2: ");
    foreach(int x in nums2)
      Console.Write(x + " ");

    Console.WriteLine();


    // Now, use copyInsert on an array of strings.
    string[] strs = { "Generics", "are", "powerful."};
    string[] strs2 = new string[4];

    // Display contents of strs.
    Console.Write("Contents of strs: ");
    foreach(string s in strs)
      Console.Write(s + " ");

    Console.WriteLine();

    // Insert into a string array.
    ArrayUtils.CopyInsert("in C#", 1, strs, strs2);

    // Display contents of strs2.
    Console.Write("Contents of strs2: ");
    foreach(string s in strs2)
      Console.Write(s + " ");
```

```
    Console.WriteLine();

    // This call is invalid because the first argument
    // is of type double, and the third and fourth arguments
    // have element types of int.
//    ArrayUtils.CopyInsert(0.01, 2, nums, nums2);
  }
}
```

The output from the program is shown here:

```
Contents of nums: 1 2 3
Contents of nums2: 1 2 99 3
Contents of strs: Generics are powerful.
Contents of strs2: Generics in C# are powerful.
```

Let's examine **CopyInsert( )** closely. First, notice how it is declared by this line:

```
public static bool CopyInsert<T>(T e, uint idx,
                                 T[] src, T[] target) {
```

The type parameter is declared *after* the method name, but *before* the parameter list. Also notice that **CopyInsert( )** is static, enabling it to be called independently of any object. Understand, though, that generic methods can be either static or non-static. There is no restriction in this regard.

Now, notice how **CopyInsert( )** is called within **Main( )** by use of the normal call syntax, without the need to specify type arguments. This is because the types of the arguments are automatically discerned, and the type of **T** is adjusted accordingly. This process is called *type inference*. For example, in the first call:

```
ArrayUtils.CopyInsert(99, 2, nums, nums2);
```

the type of **T** becomes **int** because 99 and the element types of **nums** and **nums2** are **int**. In the second call, **string** types are used, and **T** is replaced by **string**.

Now, notice the commented-out code, shown here:

```
//    ArrayUtils.CopyInsert(0.01, 2, nums, nums2);
```

If you remove the comments and then try to compile the program, you will receive an error. The reason is that the type of the first argument is **double**, but the element types of **nums** and **nums2** are **int**. However, all three types must be substituted for the same type parameter, **T.** This causes a type-mismatch, which results in a compile-time error. This ability to enforce type safety is one of the most important advantages of generic methods.

The syntax used to create **CopyInsert( )** can be generalized. Here is the general form of a generic method:

ret-type meth-name<*type-param-list*>(*param-list*) { // ...

In all cases, *type-param-list* is a comma-separated list of type parameters. Notice that for a generic method, the type parameter list follows the method name.

Using Explicit Type Arguments to Call a Generic Method

Although implicit type inference is adequate for most invocations of a generic method, it is possible to explicitly specify the type argument. To do so, specify the type argument after the method name when calling the method. For example, here **CopyInsert( )** is explicitly passed type **string**:

```
ArrayUtils.CopyInsert<string>("in C#", 1, strs, strs2);
```

You will need to explicitly specify the type when the compiler cannot infer the type for the **T** parameter or if you want to override the type inference.

Using a Constraint with a Generic Method

You can add constraints to the type arguments of a generic method by specifying them after the parameter list. For example, the following version of **CopyInsert( )** will work only with reference types:

```
public static bool CopyInsert<T>(T e, uint idx,
                                 T[] src, T[] target) where T : class {
```

If you were to try this version in the program shown earlier, then the following call to **CopyInsert( )** would not compile because **int** is a value type, not a reference type:

```
// Now wrong because T must be reference type!
ArrayUtils.CopyInsert(99, 2, nums, nums2); // Now illegal!
```

Generic Delegates

Like methods, delegates can also be generic. To declare a generic delegate, use this general form:

delegate *ret-type delegate-name*<*type-parameter-list*>(*arg-list*);

Notice the placement of the type parameter list. It immediately follows the delegate's name. The advantage of generic delegates is that they let you define, in a type-safe manner, a generalized form that can then be matched to any compatible method.

The following program demonstrates a generic delegate called **SomeOp** that has one type parameter called **T**. It returns type **T** and takes an argument of type **T**.

```
// A simple generic delegate.

using System;

// Declare a generic delegate.
delegate T SomeOp<T>(T v);

class GenDelegateDemo {
  // Return the summation of the argument.
  static int Sum(int v) {
    int result = 0;
    for(int i=v; i>0; i--)
```

```
      result += i;

    return result;
  }

  // Return a string containing the reverse of the argument.
  static string Reflect(string str) {
    string result = "";

    foreach(char ch in str)
      result = ch + result;

    return result;
  }

  static void Main() {
    // Construct an int delegate.
    SomeOp<int> intDel = Sum;
    Console.WriteLine(intDel(3));

    // Construct a string delegate.
    SomeOp<string> strDel = Reflect;
    Console.WriteLine(strDel("Hello"));
  }
}
```

The output is shown here:

```
6
olleH
```

Let's look closely at this program. First, notice how the **SomeOp** delegate is declared:

```
delegate T SomeOp<T>(T v);
```

Notice that **T** can be used as the return type even though the type parameter **T** is specified after the name **SomeOp**.

Inside **GenDelegateDemo**, the methods **Sum( )** and **Reflect( )** are declared, as shown here:

```
static int Sum(int v) {

static string Reflect(string str) {
```

The **Sum( )** method returns the summation of the integer value passed as an argument. The **Reflect( )** method returns a string that is the reverse of the string passed as an argument.

Inside **Main( )**, a delegate called **intDel** is instantiated and assigned a reference to **Sum( )**:

```
SomeOp<int> intDel = Sum;
```

Because **Sum( )** takes an **int** argument and returns an **int** value, **Sum( )** is compatible with an **int** instance of **SomeOp**.

In similar fashion, the delegate **strDel** is created and assigned a reference to **Reflect( )**:

```
SomeOp<string> strDel = Reflect;
```

Because **Reflect( )** takes a **string** argument and returns a **string** result, it is compatible with the string version of **SomeOp**.

Because of the type safety inherent in generics, you cannot assign incompatible methods to delegates. For example, assuming the preceding program, the following statement would be in error:

```
SomeOp<int> intDel = Reflect; // Error!
```

Because **Reflect( )** takes a **string** argument and returns a **string** result, it cannot be assigned to an **int** version of **SomeOp**.

As explained in Chapter 15, one of the major uses of delegates occurs when handling events. Although events, themselves, cannot be generic, the delegate that supports an event can. The following program reworks an example from Chapter 15 (the .NET-compatible event demonstration) so that it uses a generic delegate:

```
// Convert event example from Chapter 15 to use generic delegate.

using System;

// Derive a class from EventArgs.
class MyEventArgs : EventArgs {
  public int EventNum;
}

// Declare a generic delegate for an event.
delegate void MyEventHandler<T, V>(T source, V args);

// Declare an event class.
class MyEvent {
  static int count = 0;

  public event MyEventHandler<MyEvent, MyEventArgs> SomeEvent;

  // This fires SomeEvent.
  public void OnSomeEvent() {
    MyEventArgs arg = new MyEventArgs();

    if(SomeEvent != null) {
      arg.EventNum = count++;
      SomeEvent(this, arg);
    }
  }
}

class X {
  public void Handler<T, V>(T source, V arg) where V : MyEventArgs {
    Console.WriteLine("Event " + arg.EventNum +
                      " received by an X object.");
```

```
    Console.WriteLine("Source is " + source);
    Console.WriteLine();
  }
}

class Y {
  public void Handler<T,V>(T source, V arg) where V : MyEventArgs {
    Console.WriteLine("Event " + arg.EventNum +
                      " received by a Y object.");
    Console.WriteLine("Source is " + source);
    Console.WriteLine();
  }
}

class UseGenericEventDelegate {
  static void Main() {
    X ob1 = new X();
    Y ob2 = new Y();
    MyEvent evt = new MyEvent();

    // Add Handler() to the event list.
    evt.SomeEvent += ob1.Handler;
    evt.SomeEvent += ob2.Handler;

    // Fire the event.
    evt.OnSomeEvent();
    evt.OnSomeEvent();
  }
}
```

The output is show here:

```
Event 0 received by an X object.
Source is MyEvent

Event 0 received by a Y object.
Source is MyEvent

Event 1 received by an X object.
Source is MyEvent

Event 1 received by a Y object.
Source is MyEvent
```

Generic Interfaces

In addition to generic classes and methods, you can also have generic interfaces. Generic interfaces are specified just like generic classes. Here is an example that reworks the **ISeries** interface developed in Chapter 12. (Recall that **ISeries** defines the interface to a class that generates a series of numbers.) The data type upon which it operates is now specified by a type parameter.

```
// Demonstrate a generic interface.

using System;

public interface ISeries<T> {
  T GetNext(); // return next element in series
  void Reset(); // restart the series
  void SetStart(T v); // set the starting element
}

// Implement ISeries.
class ByTwos<T> : ISeries<T> {
  T start;
  T val;

  // This delegate defines the form of a method
  // that will be called when the next element in
  // the series is needed.
  public delegate T IncByTwo(T v);

  // This delegate reference will be assigned the
  // method passed to the ByTwos constructor.
  IncByTwo incr;

  public ByTwos(IncByTwo incrMeth) {
    start = default(T);
    val = default(T);
    incr = incrMeth;
  }

  public T GetNext() {
    val = incr(val);
    return val;
  }

  public void Reset() {
    val = start;
  }

  public void SetStart(T v) {
    start = v;
    val = start;
  }
}

class ThreeD {
  public int x, y, z;

  public ThreeD(int a, int b, int c) {
    x = a;
    y = b;
    z = c;
  }
}
```

```
class GenIntfDemo {
  // Define plus two for int.
  static int IntPlusTwo(int v) {
    return v + 2;
  }

  // Define plus two for double.
  static double DoublePlusTwo(double v) {
    return v + 2.0;
  }

  // Define plus two for ThreeD.
  static ThreeD ThreeDPlusTwo(ThreeD v) {
    if(v==null) return new ThreeD(0, 0, 0);
    else return new ThreeD(v.x + 2, v.y + 2, v.z + 2);
  }

  static void Main() {

    // Demonstrate int series.
    ByTwos<int> intBT = new ByTwos<int>(IntPlusTwo);

    for(int i=0; i < 5; i++)
      Console.Write(intBT.GetNext() + "  ");

    Console.WriteLine();


    // Demonstrate double series.
    ByTwos<double> dblBT = new ByTwos<double>(DoublePlusTwo);

    dblBT.SetStart(11.4);

    for(int i=0; i < 5; i++)
      Console.Write(dblBT.GetNext() + "  ");

    Console.WriteLine();


    // Demonstrate ThreeD series.
    ByTwos<ThreeD> ThrDBT = new ByTwos<ThreeD>(ThreeDPlusTwo);

    ThreeD coord;
    for(int i=0; i < 5; i++) {
      coord = ThrDBT.GetNext();
      Console.Write(coord.x + "," +
                    coord.y + "," +
                    coord.z + "  ");
    }

    Console.WriteLine();
  }
}
```

The output is shown here:

```
2  4  6  8  10
13.4  15.4  17.4  19.4  21.4
0,0,0  2,2,2  4,4,4  6,6,6  8,8,8
```

There are several things of interest in the preceding example. First, notice how **ISeries** is declared:

```
public interface ISeries<T> {
```

As mentioned, a generic interface uses a syntax similar to that of a generic class.

Now, notice how **ByTwos**, which implements **ISeries**, is declared:

```
class ByTwos<T> : ISeries<T> {
```

The type parameter **T** is specified by **ByTwos** and is also specified in **ISeries**. This is important. A class that implements a generic version of a generic interface must, itself, be generic. For example, the following declaration would be illegal because **T** is not defined:

```
class ByTwos : ISeries<T> { // Wrong!
```

The type argument required by the **ISeries** interface must be passed to **ByTwos**. Otherwise, there is no way for the interface to receive the type argument.

Next, the current value of the series, **val**, and the starting value, **start**, are declared to be objects of the generic type **T**. Then, a delegate called **IncByTwo** is declared. This delegate defines the form of a method that will be used to increase an object of type **T** by two. In order for **ByTwos** to work with any type of data, there must be some way to define what an increase by two means for each type of data. This is achieved by passing to the **ByTwos** constructor a reference to a method that performs an increase by two. This reference is stored in **incr**. When the next element in the series is needed, that method is called through the **incr** delegate to obtain the next value in the series.

Notice the class **ThreeD**. It encapsulates three-dimensional (X,Z,Y) coordinates. It is used to demonstrate **ByTwos** on a class type.

In **GenIntfDemo**, three increment methods are declared; one for **int**, one for **double**, and one for objects of type **ThreeD**. These are passed to the **ByTwos** constructor when objects of their respective types are created. Pay special attention to **ThreeDPlusTwo( )**, shown here:

```
// Define plus two for ThreeD.
static ThreeD ThreeDPlusTwo(ThreeD v) {
  if(v==null) return new ThreeD(0, 0, 0);
  else return new ThreeD(v.x + 2, v.y + 2, v.z + 2);
}
```

Notice that it first checks if **v** is **null**. If it is, then it returns a new **ThreeD** object in which all fields are set to zero. The reason for this is that **v** is set to **default(T)** by the **ByTwos** constructor. This value is zero for value types and **null** for object types. Thus, (unless **SetStart( )** has been called) for the first increment, **v** will contain **null** instead of a reference to an object. This means that for the first increment, a new object is required.

A type parameter for a generic interface can have constraints in the same way as it can for a generic class. For example, this version of **ISeries** restricts its use to reference types:

```
public interface ISeries<T> where T : class {
```

When this version of **ISeries** is implemented, the implementing class must also specify the same constraint for **T**, as shown here:

```
class ByTwos<T> : ISeries<T> where T : class {
```

Because of the reference constraint, this version of **ISeries** cannot be used on value types. Thus, in the preceding program, only **ByTwos<ThreeD>** would be valid. **ByTwos<int>** and **ByTwos<double>** would be invalid.

Comparing Instances of a Type Parameter

Sometimes you will want to compare two instances of a type parameter. For example, you might want to write a generic method called **IsIn( )** that returns true if some value is contained within an array. To accomplish this, you might first try something like this:

```
// This won't work!
public static bool IsIn<T>(T what, T[] obs) {
  foreach(T v in obs)
    if(v == what) // Error!
      return true;

  return false;
}
```

Unfortunately, this attempt won't work. Because **T** is a generic type, the compiler has no way to know precisely how two objects should be compared for equality. Should a bitwise comparison be done? Should only certain fields be compared? Should reference equality be used? The compiler has no way to answer these questions.

To enable two objects of a generic type parameter to be compared, you must use the **CompareTo( )** method defined by one of .NET's standard interfaces: **IComparable**. This interface is implemented by all of C#'s built-in types, including **int**, **string**, and **double**. It is also easy to implement for classes that you create.

The **IComparable** interface defines only the **CompareTo( )** method shown here:

int CompareTo(object *obj*)

CompareTo( ) compares the invoking object to *obj*. It returns zero if the two objects are equal, a positive value if the invoking object is greater than *obj*, and a negative value if the invoking object is less than *obj*.

To use **CompareTo( )**, you must specify a constraint that requires every type argument to implement the **IComparable** interface. Then, when you need to compare two instances of the type parameter, simply call **CompareTo( )**. For example, here is a corrected version of **IsIn( )**:

```
// Require IComparable interface.
public static bool IsIn<T>(T what, T[] obs) where T : IComparable {
  foreach(T v in obs)
```

```
    if(v.CompareTo(what) == 0) // now OK, uses CompareTo()
      return true;

  return false;
}
```

Notice the use of the constraint

```
where T : IComparable
```

This constraint ensures that only types that implement **IComparable** are valid type arguments for **IsIn( )**.

The following program demonstrates **IsIn( )**. It also shows how **IComparable** can be easily implemented by a class:

```
// Demonstrate IComparable.

using System;

class MyClass : IComparable {
  public int Val;

  public MyClass(int x) { Val = x; }

  // Implement IComparable.
  public int CompareTo(object obj) {
    return Val - ((MyClass) obj).Val;
  }
}

class CompareDemo {

  // Require IComparable interface.
  public static bool IsIn<T>(T what, T[] obs) where T : IComparable {
    foreach(T v in obs)
      if(v.CompareTo(what) == 0) // now OK, uses CompareTo()
        return true;

    return false;
  }

  // Demonstrate comparisons.
  static void Main() {
    // Use IsIn() with int.
    int[] nums = { 1, 2, 3, 4, 5 };

    if(IsIn(2, nums))
      Console.WriteLine("2 is found.");

    if(IsIn(99, nums))
      Console.WriteLine("This won't display.");


    // Use IsIn() with string.
```

```
    string[] strs = { "one", "two", "Three"};

    if(IsIn("two", strs))
      Console.WriteLine("two is found.");

    if(IsIn("five", strs))
      Console.WriteLine("This won't display.");


    // Use IsIn with MyClass.
    MyClass[] mcs = { new MyClass(1), new MyClass(2),
                      new MyClass(3), new MyClass(4) };

    if(IsIn(new MyClass(3), mcs))
      Console.WriteLine("MyClass(3) is found.");

    if(IsIn(new MyClass(99), mcs))
      Console.WriteLine("This won't display.");
  }
}
```

The output is shown here:

```
2 is found.
two is found.
MyClass(3) is found.
```

Although the preceding program is correct, there is still one potential trouble spot. Notice how **CompareTo( )** is implemented by **MyClass**:

```
public int CompareTo(object obj) {
  return Val - ((MyClass) obj).Val;
}
```

Because the parameter to **CompareTo( )** must be of type **object**, **obj** must be explicitly cast to **MyClass** in order for **Val** to be accessed. However, it's precisely this type of thing that generics were designed to eliminate!

To solve this problem, C# provides a generic version of **IComparable**, which is declared like this:

```
public interface IComparable<T>
```

In this version, the type of data being compared is passed as a type argument to **T**. This causes the declaration of **CompareTo( )** to be changed, as shown next:

```
int CompareTo(T obj)
```

Now, the parameter to **CompareTo( )** can be specified as the proper type and no cast from **object** is needed. **IComparable<T>** is also implemented by all built-in types.

Here is an improved version of **MyClass** that implements **IComparable<T>**:

```
// This version of MyClass implements IComparable<T>
class MyClass : IComparable<MyClass> {
  public int Val;
```

```
  public MyClass(int x) { Val = x; }

  public int CompareTo(MyClass obj) {
    return Val - obj.Val; // Now, no cast is needed.
  }
}
```

Notice that a cast is no longer required by this line in **CompareTo( )**:

```
return Val - obj.Val; // Now, no cast is needed.
```

Because the type parameter to **IComparable** is **MyClass**, the type of **obj** is now known to be **MyClass**.

Here is an updated version of **IsIn( )** that requires **IComparable<T>**:

```
// Require IComparable<T> interface.
public static bool IsIn<T>(T what, T[] obs) where T : IComparable<T> {
  foreach(T v in obs)
    if(v.CompareTo(what) == 0) // now OK, uses CompareTo()
      return true;

  return false;
}
```

NOTE *If a type parameter specifies a reference or a base class constraint, then* **==** *and* **!=** *can be applied to instances of that type parameter, but they only test for reference equality. To compare values, you must implement* ***IComparable*** *or* ***IComparable<T>****.*

Generic Class Hierarchies

Generic classes can be part of a class hierarchy in just the same way as non-generic classes. Thus, a generic class can act as a base class or be a derived class. The key difference between generic and non-generic hierarchies is that in a generic hierarchy, any type arguments needed by a generic base class must be passed up the hierarchy by all derived classes. This is similar to the way that constructor arguments must be passed up a hierarchy.

Using a Generic Base Class

Here is a simple example of a hierarchy that uses a generic base class:

```
// A simple generic class hierarchy.
using System;

// A generic base class.
class Gen<T> {
  T ob;

  public Gen(T o) {
    ob = o;
  }

  // Return ob.
  public T GetOb() {
```

```
    return ob;
  }
}

// A class derived from Gen.
class Gen2<T> : Gen<T> {
  public Gen2(T o) : base(o) {
    // ...
  }
}

class GenHierDemo {
  static void Main() {
    Gen2<string> g2 = new Gen2<string>("Hello");

    Console.WriteLine(g2.GetOb());
  }
}
```

In this hierarchy, **Gen2** inherits the generic class **Gen**. Notice how **Gen2** is declared by the following line:

```
class Gen2<T> : Gen<T> {
```

The type parameter **T** is specified by **Gen2** and is also passed to **Gen**. This means that whatever type is passed to **Gen2** will also be passed to **Gen**. For example, this declaration

```
Gen2<string> g2 = new Gen2<string>("Hello");
```

passes **string** as the type parameter to **Gen**. Thus, the **ob** inside the **Gen** portion of **Gen2** will be of type **string**.

Notice also that **Gen2** does not use the type parameter **T** except to pass it along to the **Gen** base class. Thus, even if a derived class would otherwise not need to be generic, it still must specify the type parameter(s) required by its generic base class.

Of course, a derived class is free to add its own type parameters, if needed. For example, here is a variation on the preceding hierarchy in which **Gen2** adds a type parameter of its own:

```
// A derived class can add its own type parameters.
using System;

// A generic base class.
class Gen<T> {
  T ob; // declare a variable of type T

  // Pass the constructor a reference of type T.
  public Gen(T o) {
    ob = o;
  }

  // Return ob.
  public T GetOb() {
    return ob;
```

```
  }
}

// A derived class of Gen that defines a second
// type parameter, called V.
class Gen2<T, V> : Gen<T> {
  V ob2;

  public Gen2(T o, V o2) : base(o) {
    ob2 = o2;
  }

  public V GetObj2() {
    return ob2;
  }
}

// Create an object of type Gen2.
class GenHierDemo2 {
  static void Main() {

    // Create a Gen2 object for string and int.
    Gen2<string, int> x =
      new Gen2<string, int>("Value is: ", 99);

    Console.Write(x.GetOb());
    Console.WriteLine(x.GetObj2());
  }
}
```

Notice the declaration of this version of **Gen2**, which is shown here:

```
class Gen2<T, V> : Gen<T> {
```

Here, **T** is the type passed to **Gen**, and **V** is the type that is specific to **Gen2**. **V** is used to declare an object called **ob2** and as a return type for the method **GetObj2( )**. In **Main( )**, a **Gen2** object is created in which type parameter **T** is **string**, and type parameter **V** is **int**. The program displays the following, expected, result:

```
Value is: 99
```

A Generic Derived Class

It is perfectly acceptable for a non-generic class to be the base class of a generic derived class. For example, consider this program:

```
// A non-generic class can be the base class of a generic derived class.
using System;

// A non-generic class.
class NonGen {
  int num;

  public NonGen(int i) {
```

```
    num = i;
  }

  public int GetNum() {
    return num;
  }
}

// A generic derived class.
class Gen<T> : NonGen {
  T ob;

  public Gen(T o, int i) : base (i) {
    ob = o;
  }

  // Return ob.
  public T GetOb() {
    return ob;
  }
}

// Create a Gen object.
class HierDemo3 {
  static void Main() {

    // Create a Gen object for string.
    Gen<String> w = new Gen<String>("Hello", 47);

    Console.Write(w.GetOb() + " ");
    Console.WriteLine(w.GetNum());
  }
}
```

The output from the program is shown here:

```
Hello 47
```

In the program, notice how **Gen** inherits **NonGen** in the following declaration:

```
class Gen<T> : NonGen {
```

Because **NonGen** is not generic, no type argument is specified. Thus, even though **Gen** declares the type parameter **T**, it is not needed by (nor can it be used by) **NonGen**. Thus, **NonGen** is inherited by **Gen** in the normal way. No special conditions apply.

Overriding Virtual Methods in a Generic Class

A virtual method in a generic class can be overridden just like any other method. For example, consider this program in which the virtual method **GetOb( )** is overridden:

```
// Overriding a virtual method in a generic class.
using System;
```

```
// A generic base class.
class Gen<T> {
  protected T ob;

  public Gen(T o) {
    ob = o;
  }

  // Return ob. This method is virtual.
  public virtual T GetOb() {
    Console.Write("Gen's GetOb(): " );
    return ob;
  }
}

// A derived class of Gen that overrides GetOb().
class Gen2<T> : Gen<T> {

  public Gen2(T o) : base(o) {  }

  // Override GetOb().
  public override T GetOb() {
    Console.Write("Gen2's GetOb(): ");
    return ob;
  }
}

// Demonstrate generic method override.
class OverrideDemo {
  static void Main() {

    // Create a Gen object for int.
    Gen<int> iOb = new Gen<int>(88);

    // This calls Gen's version of GetOb().
    Console.WriteLine(iOb.GetOb());


    // Now, create a Gen2 object and assign its
    // reference to iOb (which is a Gen<int> variable).
    iOb = new Gen2<int>(99);

    // This calls Gen2's version of GetOb().
    Console.WriteLine(iOb.GetOb());
  }
}
```

The output is shown here:

```
Gen's GetOb(): 88
Gen2's GetOb(): 99
```

As the output confirms, the overridden version of **GetOb( )** is called for an object of type **Gen2**, but the base class version is called for an object of type **Gen**.

Notice one other thing: This line

```
iOb = new Gen2<int>(99);
```

is valid because **iOb** is a variable of type **Gen<int>**. Thus, it can refer to any object of type **Gen<int>** or any object of a class derived from **Gen<int>**, including **Gen2<int>**. Of course, **iOb** couldn't be used to refer to an object of type **Gen2<double>**, for example, because of the type mismatch.

Overloading Methods That Use Type Parameters

Methods that use type parameters to declare method parameters can be overloaded. However, the rules are a bit more stringent than they are for methods that don't use type parameters. In general, a method that uses a type parameter as the data type of a parameter can be overloaded as long as the signatures of the two versions differ. This means the type and/or number of their parameters must differ. However, the determination of type difference is not based on the generic type parameter, but on the type argument substituted for the type parameter when a constructed type is created. Therefore, it is possible to overload a method that uses type parameters in such a way that it "looks right," but won't work in all specific cases.

For example, consider this generic class:

```
// Ambiguity can result when overloading methods that
// use type parameters.
//
// This program will not compile.

using System;

// A generic class that contains a potentially ambiguous
// overload of the Set() method.
class Gen<T, V> {
  T ob1;
  V ob2;

  // ...

  // In some cases, these two methods
  // will not differ in their parameter types.
  public void Set(T o) {
    ob1 = o;
  }

  public void Set(V o) {
    ob2 = o;
  }
}

class AmbiguityDemo {
  static void Main() {
    Gen<int, double> ok = new Gen<int, double>();
```

```
    Gen<int, int> notOK = new Gen<int, int>();

    ok.Set(10); // is valid, type args differ

    notOK.Set(10); // ambiguous, type args are the same!
  }
}
```

Let's examine this program closely. First, notice that **Gen** declares two type parameters: **T** and **V**. Inside **Gen**, **Set( )** is overloaded based on parameters of type **T** and **V**, as shown here:

```
public void Set(T o) {
  ob1 = o;
}

public void Set(V o) {
  ob2 = o;
}
```

This looks reasonable because **T** and **V** appear to be different types. However, this overloading creates a potential ambiguity problem.

As **Gen** is written, there is no requirement that **T** and **V** actually be different types. For example, it is perfectly correct (in principle) to construct a **Gen** object as shown here:

```
Gen<int, int> notOK = new Gen<int, int>();
```

In this case, both **T** and **V** will be replaced by **int**. This makes both versions of **Set( )** identical, which is, of course, an error. Thus, when the attempt to call **Set( )** on **notOK** occurs later in **Main( )**, a compile-time ambiguity error is reported.

In general, you can overload methods that use type parameters as long as there is no constructed type that results in a conflict. It is important to understand that type constraints do not participate in overload resolution. Thus, type constraints cannot be used to eliminate ambiguity. Like methods, constructors, operators, and indexers that use type parameters can also be overloaded, and the same rules apply.

How Generic Types Are Instantiated

One question that is often raised when working with generics is whether the use of a generic class leads to code-bloat at runtime. The simple answer is no. The reason is that C# implements generics in a highly efficient manner that creates new constructed types only when they are needed. Here is how the process works.

When a generic class is compiled into MSIL, it retains all of its type parameters in their generic form. At runtime, when a specific instance of the class is required, the JIT compiler constructs a specific, executable code version of the class in which the type parameters are replaced by the type arguments. Each instance of the class that uses the same type arguments will use the same executable code version.

For example, given some generic class called **Gen<T>**, then all **Gen<int>** objects will use the same executable code. Thus, code-bloat is reduced and only those versions of the class that are actually used in the program will be created. When a different constructed type is needed, a new version of the class is compiled.

In general, a new executable version of a generic class is created for each constructed type in which the type argument is a value type, such as **int** or **double**. Thus, each object of **Gen<int>** will use one version of **Gen** and each object of type **Gen<double>** will use another version of **Gen**, with each version of **Gen** tailored to the specific value type. However, there will be *only one version* of a generic class that handles all cases in which the type argument is a reference type. This is because the size (in bytes) of all references is the same. Thus, only one version is needed to handle all types of references. This optimization also reduces code-bloat.

Some Generic Restrictions

Here are a few restrictions that you need to keep in mind when using generics:

- Properties, operators, indexers, and events cannot be generic. However, these items can be used in a generic class and can make use of the generic type parameters of that class.
- The **extern** modifier cannot be applied to a generic method.
- Pointer types cannot be used as type arguments.
- If a generic class contains a **static** field, then *each constructed type has its own copy* of that field. This means that each instance of the *same constructed type* shares the same **static** field. However, a different constructed type shares a different copy of that field. Thus, a **static** field is not shared by all constructed types.

Final Thoughts on Generics

Generics are a powerful extension to C# because they streamline the creation of type-safe, reusable code. Although the generic syntax can seem a bit overwhelming at first, it will quickly become second nature. Likewise, learning how and when to use constraints takes a bit of practice, but becomes easier over time. Generics are now an integral part of C# programming. It's worth the effort it takes to master this important feature.

19 CHAPTER

LINQ

Future generations of programmers will look back on version 3.0 as a pivotal event in the evolution of C# because it fundamentally and irrevocably reshapes the core of the language. The reason for this dramatic impact can be stated in a single acronym: *LINQ*. LINQ adds to C# an entirely new syntactic element, several new keywords, and a powerful new capability. The inclusion of LINQ significantly increases the scope of the language, expanding the range of tasks to which C# can be applied. Moreover, LINQ has charted the future direction of computer language development because it offers a new way to think about and solve some of the most common, yet challenging problems that face today's programmers. Simply put, the integration of LINQ into C# sets a new standard that will affect the course of language design well into the future. LINQ *is* that important.

NOTE *LINQ in C# is essentially a language within a language. As a result, the subject of LINQ is quite large, involving many features, options, and alternatives. Although this chapter describes LINQ in significant detail, it is not possible to explore all facets, nuances, and applications of this powerful feature. To do so would require an entire book of its own. Instead, this chapter focuses on the core elements of LINQ and presents numerous examples. It is important to understand that we are just now at the beginning of the LINQ revolution. Going forward, LINQ is a subsystem that you will want to study in greater detail.*

What Is LINQ?

LINQ stands for *Language-Integrated Query.* It encompasses a set of features that let you retrieve information from a data source. As you may know, the retrieval of data constitutes an important part of many programs. For example, a program might obtain information from a customer list, look up product information in a catalog, or access an employee's record. In many cases, such data is stored in a database that is separate from the application. For example, a product catalog might be stored in a relational database. In the past, interacting with such a database would involve generating queries using Structured Query Language (SQL). Other sources of data, such as XML, required their own approach. Therefore, prior to C# 3.0, support for such queries was not built into C#. LINQ changes this.

LINQ adds to C# the ability to generate queries for any LINQ-compatible data source. Furthermore, the syntax used for the query is the same, no matter what data source is used. This means that the syntax used to query data in a relational database is the same as that

used to query data stored in an array, for example. It is no longer necessary to use SQL or any other non-C# mechanism. The query capability is fully integrated into the C# language.

In addition to using LINQ with SQL, LINQ can be used with XML files and ADO.NET Datasets. Perhaps equally important, it can also be used with C# arrays and collections (described in Chapter 24). Therefore, LINQ gives you a uniform way to access data. This is a powerful, innovative concept. It is not only changing the way that data is accessed, but it also offers a new way to think about and approach old problems. In the future, many programming solutions will be crafted in terms of LINQ. Its effects will not be limited to just database access.

LINQ is supported by a set of interrelated features, including the query syntax added to the C# language, lambda expressions, anonymous types, and extension methods. Lambda expressions are described in Chapter 15. The others are examined here.

LINQ Fundamentals

At LINQ's core is the *query*. A query specifies what data will be obtained from a data source. For example, a query on a customer mailing list might request the addresses of all customers that reside in a specific city, such as Chicago or Tokyo. A query on an inventory database might request a list of out-of-stock items. A query on a log of Internet usage could ask for a list of the websites with the highest hit counts. Although these queries differ in their specifics, all can be expressed using the same LINQ syntactic elements.

After a query has been created, it can be executed. One way this is done is by using the query in a **foreach** loop. Executing a query causes its results to be obtained. Thus, using a query involves two key steps. First, the form of the query is created. Second, the query is executed. Therefore, the query defines *what* to retrieve from a data source. Executing the query actually *obtains the results.*

In order for a source of data to be used by LINQ, it must implement the **IEnumerable** interface. There are two forms of this interface: one generic, one not. In general, it is easier if the data source implements the generic version, **IEnumerable<T>**, where **T** specifies the type of data being enumerated. The rest of the chapter assumes that a data source implements **IEnumerable<T>**. This interface is declared in **System.Collections.Generic**. A class that implements **IEnumerable<T>** supports enumeration, which means that its contents can be obtained one at a time, in sequence. All C# arrays support **IEnumerable<T>**. Thus, arrays can be used to demonstrate the central concepts of LINQ. Understand, however, that LINQ is not limited to arrays.

A Simple Query

At this point, it will be helpful to work through a simple LINQ example. The following program uses a query to obtain the positive values contained in an array of integers:

```
// Create a simple LINQ query.
using System;
using System.Linq;

class SimpQuery {
  static void Main() {

    int[] nums =  { 1, -2, 3, 0, -4, 5 };
```

```
    // Create a query that obtains only positive numbers.
    var posNums = from n in nums
                  where n > 0
                  select n;

    Console.Write("The positive values in nums: ");

    // Execute the query and display the results.
    foreach(int i in posNums) Console.Write(i + " ");

    Console.WriteLine();
  }
}
```

This program produces the following output:

```
The positive values in nums: 1 3 5
```

As you can see, only the positive values in the **nums** array are displayed. Although quite simple, this program demonstrates the key features of LINQ. Let's examine it closely.

The first thing to notice in the program is the **using** directive:

```
using System.Linq;
```

To use the LINQ features, you must include the **System.Linq** namespace.

Next, an array of **int** called **nums** is declared. All arrays in C# are implicitly convertible to **IEnumerable<T>**. This makes any C# array usable as a LINQ data source.

Next, a query is declared that retrieves those elements in **nums** that are positive. It is shown here:

```
var posNums = from n in nums
              where n > 0
              select n;
```

The variable **posNums** is called the *query variable.* It refers to the set of rules defined by the query. Notice it uses **var** to implicitly declare **posNums**. As you know, this makes **posNums** an implicitly typed variable. In queries, it is often convenient to use implicitly typed variables, although you can also explicitly declare the type (which must be some form of **IEnumerable<T>**). The variable **posNums** is then assigned the query expression.

All queries begin with **from**. This clause specifies two items. The first is the *range variable,* which will receive elements obtained from the data source. In this case, the range variable is **n**. The second item is the data source, which in this case is the **nums** array. The type of the range variable is inferred from the data source. In this case, the type of **n** is **int**. Generalizing, here is the syntax of the **from** clause:

from *range-variable* in *data-source*

The next clause in the query is **where**. It specifies a condition that an element in the data source must meet in order to be obtained by the query. Its general form is shown here:

where *boolean-expression*

The *boolean-expression* must produce a **bool** result. (This expression is also called a *predicate*.) There can be more than one **where** clause in a query. In the program, this **where** clause is used:

```
where n > 0
```

It will be true only for an element whose value is greater than zero. This expression will be evaluated for every **n** in **nums** when the query executes. Only those values that satisfy this condition will be obtained. In other words, a **where** clause acts as a filter on the data source, allowing only certain items through.

All queries end with either a **select** clause or a **group** clause. This example employs the **select** clause. It specifies precisely what is obtained by the query. For simple queries, such as the one in this example, the range value is selected. Therefore, it returns those integers from **nums** that satisfy the **where** clause. In more sophisticated situations, it is possible to finely tune what is selected. For example, when querying a mailing list, you might return just the last name of each recipient, rather than the entire address. Notice that the **select** clause ends with a semicolon. Because **select** ends a query, it ends the statement and requires a semicolon. Notice, however, that the other clauses in the query do not end with a semicolon.

At this point, a query variable called **posNums** has been created, but no results have been obtained. It is important to understand that a query simply defines a set of rules. It is not until the query is executed that results are obtained. Furthermore, the same query can be executed two or more times, with the possibility of differing results if the underlying data source changes between executions. Therefore, simply declaring the query **posNums** does not mean that it contains the results of the query.

To execute the query, the program uses the **foreach** loop shown here:

```
foreach(int i in posNums) Console.WriteLine(i + " ");
```

Notice that **posNums** is specified as the collection being iterated over. When the **foreach** executes, the rules defined by the query specified by **posNums** are executed. With each pass through the loop, the next element returned by the query is obtained. The process ends when there are no more elements to retrieve. In this case, the type of the iteration variable **i** is explicitly specified as **int** because this is the type of the elements retrieved by the query. Explicitly specifying the type of the iteration variable is fine in this situation, since it is easy to know the type of the value selected by the query. However, in more complicated situations, it will be easier (or in some cases, necessary) to implicitly specify the type of the iteration variable by using **var**.

A Query Can Be Executed More Than Once

Because a query defines a set of rules that are used to retrieve data, but does not, itself, produce results, the same query can be run multiple times. If the data source changes between runs, then the results of the query may differ. Therefore, once you define a query, executing it will always produce the most current results. Here is an example. In the following version of the preceding program, the contents of the **nums** array are changed between two executions of **posNums**:

```
// Create a simple query.
using System;
using System.Linq;
using System.Collections.Generic;
```

```
class SimpQuery {
  static void Main() {

    int[] nums =  { 1, -2, 3, 0, -4, 5 };

    // Create a query that obtains only positive numbers.
    var posNums = from n in nums
                  where n > 0
                  select n;

    Console.Write("The positive values in nums: ");

    // Execute the query and display the results.
    foreach(int i in posNums) Console.Write(i + " ");
    Console.WriteLine();

    // Change nums.
    Console.WriteLine("\nSetting nums[1] to 99.");
    nums[1] = 99;

    Console.Write("The positive values in nums after change: ");

    // Execute the query a second time.
    foreach(int i in posNums) Console.Write(i + " ");
    Console.WriteLine();
  }
}
```

The following output is produced:

```
The positive values in nums: 1 3 5

Setting nums[1] to 99.
The positive values in nums after change: 1 99 3 5
```

As the output confirms, after the value in **nums[1]** was changed from –2 to 99, the result of rerunning the query reflects the change. This is a key point that must be emphasized. Each execution of a query produces its own results, which are obtained by enumerating the current contents of the data source. Therefore, if the data source changes, so, too, might the results of executing a query. The benefits of this approach are quite significant. For example, if you are obtaining a list of pending orders for an online store, then you want each execution of your query to produce all orders, including those just entered.

How the Data Types in a Query Relate

As the preceding examples have shown, a query involves variables whose types relate to one another. These are the query variable, the range variable, and the data source. Because the correspondence among these types is both important and a bit confusing at first, they merit a closer look.

The type of the range variable must agree with the type of the elements stored in the data source. Thus, the type of the range variable is dependent upon the type of the data source. In many cases, C# can infer the type of the range variable. As long as the data source implements **IEnumerable<T>**, the type inference can be made because **T** describes the type

of the elements in the data source. (As mentioned, all arrays implement **IEnumerable<T>**, as do many other data sources.) However, if the data source implements the non-generic version of **IEnumerable**, then you will need to explicitly specify the type of the range variable. This is done by specifying its type in the **from** clause. For example, assuming the preceding examples, this shows how to explicitly declare **n** to be an **int**:

```
var posNums = from int n in nums
  // ...
```

Of course, the explicit type specification is not needed here because all arrays are implicitly convertible to **IEnumerable<T>**, which enables the type of the range variable to be inferred.

The type of object returned by a query is an instance of **IEnumerable<T>**, where **T** is the type of the elements. Thus, the type of the query variable must be an instance of **IEnumerable<T>**. The value of **T** is determined by the type of the value specified by the **select** clause. In the case of the preceding examples, **T** is **int** because **n** is an **int**. (As explained, **n** is an **int** because **int** is the type of elements stored in **nums**.) Therefore, the query could have been written like this, with the type explicitly specified as **IEnumerable <int>**:

```
IEnumerable<int> posNums = from n in nums
                           where n > 0
                           select n;
```

The key point is that the type of the item selected by **select** must agree with the type argument passed to **IEnumerable<T>** used to declare the query variable. Often query variables use **var** rather than explicitly specifying the type because this lets the compiler infer the proper type from the **select** clause. As you will see, this approach is particularly useful when **select** returns something other than an individual element from the data source.

When a query is executed by the **foreach** loop, the type of the iteration variable must be the same as the type of the range variable. In the preceding examples, this type was explicitly specified as **int**, but you can let the compiler infer the type by specifying this variable as **var**. As you will see, there are also some cases in which **var** must be used because the data type has no name.

The General Form of a Query

All queries share a general form, which is based on a set of contextual keywords, shown here:

ascending	descending	equals	from
group	in	into	join
let	on	orderby	select
where			

Of these, the following begin query clauses:

from	group	join	let
orderby	select	where	

A query must begin with the keyword **from** and end with either a **select** or **group** clause. The **select** clause determines what type of value is enumerated by the query. The **group**

clause returns the data by groups, with each group being able to be enumerated individually. As the preceding examples have shown, the **where** clause specifies criteria that an item must meet in order for it to be returned. The remaining clauses help you fine-tune a query. The follows sections examine each query clause.

Filter Values with where

As explained, **where** is used to filter the data returned by a query. The preceding examples have shown only its simplest form, in which a single condition is used. A key point to understand is that you can use **where** to filter data based on more than one condition. One way to do this is through the use of multiple **where** clauses. For example, consider the following program that displays only those values in the array that are both positive and less than 10:

```
// Use multiple where clauses.
using System;
using System.Linq;

class TwoWheres {
  static void Main() {

    int[] nums =  { 1, -2, 3, -3, 0, -8, 12, 19, 6, 9, 10 };

    // Create a query that obtains positive values less than 10.
    var posNums = from n in nums
                  where n > 0
                  where n < 10
                  select n;

    Console.Write("The positive values less than 10: ");

    // Execute the query and display the results.
    foreach(int i in posNums) Console.Write (i + " ");
    Console.WriteLine();
  }
}
```

The output is shown here:

```
The positive values less than 10: 1 3 6 9
```

As you can see, only positive values less than 10 are retrieved. This outcome is achieved by the use of the following two **where** clauses:

```
where n > 0
where n < 10
```

The first **where** requires that an element be greater than zero. The second requires the element to be less than 10. Thus, an element must be between 1 and 9 (inclusive) to satisfy both clauses.

Although it is not wrong to use two **where** clauses as just shown, the same effect can be achieved in a more compact manner by using a single **where** in which both tests are combined into a single expression. Here is the query rewritten to use this approach:

```
var posNums = from n in nums
              where n > 0 && n < 10
              select n;
```

In general, a **where** condition can use any valid C# expression that evaluates to a Boolean result. For example, the following program defines an array of **string**s. Several of the strings define Internet addresses. The query **netAddrs** retrieves only those strings that have more than four characters and that end with ".net". Thus, it finds those strings that contain Internet addresses that use the **.net** top-level domain name.

```
// Demonstrate another where clause.
using System;
using System.Linq;

class WhereDemo2 {

  static void Main() {

    string[] strs = { ".com", ".net", "hsNameA.com", "hsNameB.net",
                      "test", ".network", "hsNameC.net", "hsNameD.com" };

    // Create a query that obtains Internet addresses that
    // end with .net.
    var netAddrs = from addr in strs
                   where addr.Length > 4 && addr.EndsWith(".net")
                   select addr;

    // Execute the query and display the results.
    foreach(var str in netAddrs) Console.WriteLine(str);
  }
}
```

The output is shown here:

```
hsNameB.net
hsNameC.net
```

Notice that the program makes use of one of **string**'s methods called **EndsWith( )** within the **where** clause. It returns true if the invoking string ends with the character sequence specified as an argument.

Sort Results with orderby

Often you will want the results of a query to be sorted. For example, you might want to obtain a list of past-due accounts, in order of the remaining balance, from greatest to least. Or, you might want to obtain a customer list, alphabetized by name. Whatever the purpose, LINQ gives you an easy way to produce sorted results: the **orderby** clause.

You can use **orderby** to sort on one or more criteria. We will begin with the simplest case: sorting on a single item. The general form of **orderby** that sorts based on a single criterion is shown here:

orderby *sort-on how*

The item on which to sort is specified by *sort-on.* This can be as inclusive as the entire element stored in the data source or as restricted as a portion of a single field within the element. The value of *how* determines if the sort is ascending or descending, and it must be either **ascending** or **descending**. The default direction is ascending, so you won't normally specify **ascending**.

Here is an example that uses **orderby** to retrieve the values in an **int** array in ascending order:

```
// Demonstrate orderby.
using System;
using System.Linq;

class OrderbyDemo {

  static void Main() {

    int[] nums =  { 10, -19, 4, 7, 2, -5, 0 };

    // Create a query that obtains the values in sorted order.
    var posNums = from n in nums
                  orderby n
                  select n;

    Console.Write("Values in ascending order: ");

    // Execute the query and display the results.
    foreach(int i in posNums) Console.Write(i + " ");

    Console.WriteLine();
  }
}
```

The output is shown here:

```
Values in ascending order: -19 -5 0 2 4 7 10
```

To change the order to descending, simply specify the **descending** option, as shown here:

```
var posNums = from n in nums
              orderby n descending
              select n;
```

If you try this, you will see that the order of the values is reversed.

Although sorting on a single criterion is often what is needed, you can use **orderby** to sort on multiple items by using this form:

orderby *sort-onA direction*, *sort-onB direction*, *sort-onC direction*, …

In this form, *sort-onA* is the item on which the primary sorting is done. Then, each group of equivalent items is sorted on *sort-onB*, and each of those groups is sorted on *sort-onC*, and so on. Thus, each subsequent *sort-on* specifies a "then by" item on which to sort. In all cases, *direction* is optional, defaulting to **ascending**. Here is an example that uses three sort criteria to sort bank account information by last name, then by first name, and finally by account balance:

```
// Sort on multiple criteria with orderby.
using System;
using System.Linq;

class Account {
  public string FirstName { get; private set; }
  public string LastName { get; private set; }
  public double Balance { get; private set; }
  public string AccountNumber { get; private set; }

  public Account(string fn, string ln, string accnum, double b) {
    FirstName = fn;
    LastName = ln;
    AccountNumber = accnum;
    Balance = b;
  }
}

class OrderbyDemo {

  static void Main() {

    // Create some data.
    Account[] accounts = { new Account("Tom", "Smith", "132CK", 100.23),
                           new Account("Tom", "Smith", "132CD", 10000.00),
                           new Account("Ralph", "Jones", "436CD", 1923.85),
                           new Account("Ralph", "Jones", "454MM", 987.132),
                           new Account("Ted", "Krammer", "897CD", 3223.19),
                           new Account("Ralph", "Jones", "434CK", -123.32),
                           new Account("Sara", "Smith", "543MM", 5017.40),
                           new Account("Sara", "Smith", "547CD", 34955.79),
                           new Account("Sara", "Smith", "843CK", 345.00),
                           new Account("Albert", "Smith", "445CK", 213.67),
                           new Account("Betty", "Krammer","968MM",5146.67),
                           new Account("Carl", "Smith", "078CD", 15345.99),
                           new Account("Jenny", "Jones", "108CK", 10.98)
                         };

    // Create a query that obtains the accounts in sorted order.
    // Sorting first by last name, then within same last names sorting by
    // by first name, and finally by account balance.
    var accInfo = from acc in accounts
                  orderby acc.LastName, acc.FirstName, acc.Balance
                  select acc;
```

```
    Console.WriteLine("Accounts in sorted order: ");

    string str = "";

    // Execute the query and display the results.
    foreach(Account acc in accInfo) {
      if(str != acc.FirstName) {
        Console.WriteLine();
        str = acc.FirstName;
      }

      Console.WriteLine("{0}, {1}\tAcc#: {2}, {3,10:C}",
                        acc.LastName, acc.FirstName,
                        acc.AccountNumber, acc.Balance);
    }
    Console.WriteLine();
  }
}
```

The output is shown here:

```
Accounts in sorted order:

Jones, Jenny       Acc#: 108CK,     $10.98

Jones, Ralph       Acc#: 434CK,   ($123.32)
Jones, Ralph       Acc#: 454MM,    $987.13
Jones, Ralph       Acc#: 436CD,  $1,923.85

Krammer, Betty     Acc#: 968MM,  $5,146.67

Krammer, Ted       Acc#: 897CD,  $3,223.19

Smith, Albert      Acc#: 445CK,   ($213.67)

Smith, Carl        Acc#: 078CD, $15,345.99

Smith, Sara        Acc#: 843CK,    $345.00
Smith, Sara        Acc#: 543MM,  $5,017.40
Smith, Sara        Acc#: 547CD, $34,955.79

Smith, Tom         Acc#: 132CK,    $100.23
Smith, Tom         Acc#: 132CD, $10,000.00
```

In the query, look closely at how the **orderby** clause is written:

```
var accInfo = from acc in accounts
              orderby acc.LastName, acc.FirstName, acc.Balance
              select acc;
```

Here is how it works. First, the results are sorted by last name, and then entries with the same last name are sorted by the first name. Finally, groups of entries with the same first

and last name are sorted by the account balance. This is why the list of accounts under the name Jones is shown in this order:

```
Jones, Jenny       Acc#: 108CK,     $10.98

Jones, Ralph       Acc#: 434CK,   ($123.32)
Jones, Ralph       Acc#: 454MM,     $987.13
Jones, Ralph       Acc#: 436CD,   $1,923.85
```

As the output confirms, the list is sorted by last name, then by first name, and finally by account balance.

When using multiple criteria, you can reverse the condition of any sort by applying the **descending** option. For example, this query causes the results to be shown in order of decreasing balance:

```
var accInfo = from acc in accounts
              orderby x.LastName, x.FirstName, x.Balance descending
              select acc;
```

When using this version, the list of Jones entries will be displayed like this:

```
Jones, Jenny       Acc#: 108CK,     $10.98

Jones, Ralph       Acc#: 436CD,   $1,923.85
Jones, Ralph       Acc#: 454MM,     $987.13
Jones, Ralph       Acc#: 434CK,   ($123.32)
```

As you can see, now the accounts for Ralph Jones are displayed from greatest to least.

A Closer Look at select

The **select** clause determines what type of elements are obtained by a query. Its general form is shown here:

select *expression*

So far we have been using **select** to return the range variable. Thus, *expression* has simply named the range variable. However, **select** is not limited to this simple action. It can return a specific portion of the range variable, the result of applying some operation or transformation to the range variable, or even a new type of object that is constructed from pieces of the information retrieved from the range variable. This is called *projecting*.

To begin examining the other capabilities of **select**, consider the following program. It displays the square roots of the positive values contained in an array of **double** values.

```
// Use select to return the square root of all positive values
// in an array of doubles.
using System;
using System.Linq;

class SelectDemo {

  static void Main() {
```

```
    double[] nums =  { -10.0, 16.4, 12.125, 100.85, -2.2, 25.25, -3.5 } ;

    // Create a query that returns the square roots of the
    // positive values in nums.
    var sqrRoots = from n in nums
                   where n > 0
                   select Math.Sqrt(n);

    Console.WriteLine("The square roots of the positive values" +
                      " rounded to two decimal places:");

    // Execute the query and display the results.
    foreach(double r in sqrRoots) Console.WriteLine("{0:#.##}", r);
  }
}
```

The output is shown here:

```
The square roots of the positive values rounded to two decimal places:
4.05
3.48
10.04
5.02
```

In the query, pay special attention to the **select** clause:

```
select Math.Sqrt(n);
```

It returns the square root of the range variable. It does this by obtaining the result of passing the range variable to **Math.Sqrt( )**, which returns the square root of its argument. This means that the sequence obtained when the query is executed will contain the square roots of the positive values in **nums**. If you generalize this concept, the power of **select** becomes apparent. You can use **select** to generate any type of sequence you need, based on the values obtained from the data source.

Here is a program that shows another way to use **select**. It creates a class called **EmailAddress** that contains two properties. The first holds a person's name. The second contains an e-mail address. The program then creates an array that contains several **EmailAddress** entries. The program uses a query to obtain a list of just the e-mail addresses by themselves.

```
// Return a portion of the range variable.
using System;
using System.Linq;

class EmailAddress {
  public string Name { get; set; }
  public string Address { get; set; }

  public EmailAddress(string n, string a) {
    Name = n;
    Address = a;
  }
}
```

```
class SelectDemo2 {
  static void Main() {

    EmailAddress[] addrs = {
         new EmailAddress("Herb", "Herb@HerbSchildt.com"),
         new EmailAddress("Tom", "Tom@HerbSchildt.com"),
         new EmailAddress("Sara", "Sara@HerbSchildt.com")
    };

    // Create a query that selects e-mail addresses.
    var eAddrs = from entry in addrs
                 select entry.Address;

    Console.WriteLine("The e-mail addresses are");

    // Execute the query and display the results.
    foreach(string s in eAddrs) Console.WriteLine("  " + s);
  }
}
```

The output is shown here:

```
The e-mail addresses are
  Herb@HerbSchildt.com
  Tom@HerbSchildt.com
  Sara@HerbSchildt.com
```

Pay special attention to the **select** clause:

```
select entry.Address;
```

Instead of returning the entire range variable, it returns only the **Address** portion. This fact is evidenced by the output. This means the query returns a sequence of strings, not a sequence of **EmailAddress** objects. This is why the **foreach** loop specifies **s** as a **string**. As explained, the type of sequence returned by a query is determined by the type of value returned by the **select** clause.

One of the more powerful features of **select** is its ability to return a sequence that contains elements created during the execution of the query. For example, consider the following program. It defines a class called **ContactInfo**, which stores a name, e-mail address, and telephone number. It also defines the **EmailAddress** class used by the preceding example. Inside **Main( )**, an array of **ContactInfo** is created. Then, a query is declared in which the data source is an array of **ContactInfo**, but the sequence returned contains **EmailAddress** objects. Thus, the type of the sequence returned by **select** is not **ContactInfo**, but rather **EmailAddress**, and these objects are created during the execution of the query.

```
// Use a query to obtain a sequence of EmailAddresses
// from a list of ContactInfo.
using System;
using System.Linq;

class ContactInfo {
  public string Name { get; set; }
  public string Email { get; set; }
```

```
  public string Phone { get; set; }

  public ContactInfo(string n, string a, string p) {
    Name = n;
    Email = a;
    Phone = p;
  }
}

class EmailAddress {
  public string Name { get; set; }
  public string Address { get; set; }

  public EmailAddress(string n, string a) {
    Name = n;
    Address = a;
  }
}

class SelectDemo3 {
  static void Main() {

    ContactInfo[] contacts = {
         new ContactInfo("Herb", "Herb@HerbSchildt.com", "555-1010"),
         new ContactInfo("Tom", "Tom@HerbSchildt.com", "555-1101"),
         new ContactInfo("Sara", "Sara@HerbSchildt.com", "555-0110")
    };

    // Create a query that creates a list of EmailAddress objects.
    var emailList = from entry in contacts
                    select new EmailAddress(entry.Name, entry.Email);

    Console.WriteLine("The e-mail list is");

    // Execute the query and display the results.
    foreach(EmailAddress e in emailList)
      Console.WriteLine("  {0}: {1}", e.Name, e.Address );
  }
}
```

The output is shown here:

```
The e-mail list is
  Herb: Herb@HerbSchildt.com
  Tom: Tom@HerbSchildt.com
  Sara: Sara@HerbSchildt.com
```

In the query, pay special attention to the **select** clause :

```
select new EmailAddress(entry.Name, entry.Email);
```

It creates a new **EmailAddress** object that contains the name and e-mail address obtained from a **ContactInfo** object in the **contacts** array. The key point is that new **EmailAddress** objects are created by the query in its **select** clause, during the query's execution.

Use Nested from Clauses

A query can contain more than one **from** clause. Thus, a query can contain nested **from** clauses. One common use of a nested **from** clause is found when a query needs to obtain data from two different sources. Here is a simple example. It uses two **from** clauses to iterate over two different character arrays. It produces a sequence that contains all possible combinations of the two sets of characters.

```
// Use two from clauses to create a list of all
// possible combinations of the letters A, B, and C
// with the letters X, Y, and Z.
using System;
using System.Linq;


// This class holds the result of the query.
class ChrPair {
  public char First;
  public char Second;

  public ChrPair(char c, char c2) {
    First = c;
    Second = c2;
  }
}

class MultipleFroms {
  static void Main() {

    char[] chrs = { 'A', 'B', 'C' };
    char[] chrs2 = { 'X', 'Y', 'Z' };

    // Notice that the first from iterates over chrs and
    // the second from iterates over chrs2.
    var pairs = from ch1 in chrs
                from ch2 in chrs2
                select new ChrPair(ch1, ch2);

    Console.WriteLine("All combinations of ABC with XYZ: ");

    foreach(var p in pairs)
      Console.WriteLine("{0} {1}", p.First, p.Second);
  }
}
```

The output is shown here:

```
All combinations of ABC with XYZ:
A X
A Y
A Z
B X
B Y
```

```
B Z
C X
C Y
C Z
```

The program begins by creating a class called **ChrPair** that will hold the results of the query. It then creates two character arrays, called **chrs** and **chrs2**. It uses the following query to produce all possible combinations of the two sequences:

```
var pairs = from ch1 in chrs
            from ch2 in chrs2
            select new ChrPair(ch1, ch2);
```

The nested **from** clauses cause both **chrs** and **chrs2** to be iterated over. Here is how it works. First, a character is obtained from **chrs** and stored in **ch1**. Then, the **chrs2** array is enumerated. With each iteration of the inner **from**, a character from **chrs2** is stored in **ch2** and the **select** clause is executed. The result of the **select** clause is a new object of type **ChrPair** that contains the character pair **ch1**, **ch2** produced by each iteration of the inner **from**. Thus, a **ChrPair** is produced in which each possible combination of characters is obtained.

Another common use of a nested **from** is to iterate over a data source that is contained within another data source. An example of this is found in the section, "Use **let** to Create a Variable in a Query," later in this chapter.

Group Results with group

One of the most powerful query features is provided by the **group** clause because it enables you to create results that are grouped by keys. Using the sequence obtained from a group, you can easily access all of the data associated with a key. This makes **group** an easy and effective way to retrieve data that is organized into sequences of related items. The **group** clause is one of only two clauses that can end a query. (The other is **select**.)

The **group** clause has the following general form:

group *range-variable* by *key*

It returns data grouped into sequences, with each sequence sharing the key specified by *key*.

The result of **group** is a sequence that contains elements of type **IGrouping<TKey, TElement>**, which is declared in the **System.Linq** namespace. It defines a collection of objects that share a common key. The type of query variable in a query that returns a group is **IEnumerable<IGrouping<TKey, TElement>>**. **IGrouping** defines a read-only property called **Key**, which returns the key associated with each sequence.

Here is an example that illustrates the use of **group**. It declares an array that contains a list of websites. It then creates a query that groups the list by top-level domain name, such as **.org** or **.com**.

```
// Demonstrate the group clause.
using System;
using System.Linq;

class GroupDemo {

  static void Main() {
```

```
    string[] websites = { "hsNameA.com", "hsNameB.net", "hsNameC.net",
                          "hsNameD.com", "hsNameE.org", "hsNameF.org",
                          "hsNameG.tv",  "hsNameH.net", "hsNameI.tv" };

    // Create a query that groups websites by top-level domain name.
    var webAddrs = from addr in websites
                   where addr.LastIndexOf(".") != -1
                   group addr by addr.Substring(addr.LastIndexOf("."));

    // Execute the query and display the results.
    foreach(var sites in webAddrs) {
      Console.WriteLine("Web sites grouped by " + sites.Key);
      foreach(var site in sites)
        Console.WriteLine("  " + site);
      Console.WriteLine();
    }
  }
}
```

The output is shown here:

```
Web sites grouped by .com
  hsNameA.com
  hsNameD.com

Web sites grouped by .net
  hsNameB.net
  hsNameC.net
  hsNameH.net

Web sites grouped by .org
  hsNameE.org
  hsNameF.org

Web sites grouped by .tv
  hsNameG.tv
  hsNameI.tv
```

As the output shows, the data is grouped based on the top-level domain name of a website. Notice how this is achieved by the **group** clause:

```
var webAddrs = from addr in websites
               where addr.LastIndexOf(".") != -1
               group addr by addr.Substring(addr.LastIndexOf("."));
```

The key is obtained by use of the **LastIndexOf( )** and **Substring( )** methods defined by **string**. (These are described in Chapter 7. The version of **Substring( )** used here returns the substring that starts at the specified index and runs to the end of the invoking string.) The index of the last period in a website name is found using **LastIndexOf( )**. Using this index, the **Substring( )** method obtains the remainder of the string, which is the part of the website name that contains the top-level domain name. One other point: Notice the use of the **where** clause to filter out any strings that don't contain a period. The **LastIndexOf( )** method returns –1 if the specified string is not contained in the invoking string.

Because the sequence obtained when **webAddrs** is executed is a list of groups, you will need to use two **foreach** loops to access the members of each group. The outer loop obtains each group. The inner loop enumerates the members within the group. The iteration variable of the outer **foreach** loop must be an **IGrouping** instance compatible with the key and element type. In the example both the keys and elements are **string**. Therefore, the type of the **sites** iteration variable of the outer loop is **IGrouping<string, string>**. The type of the iteration variable of the inner loop is **string**. For brevity, the example implicitly declares these variables, but they could have been explicitly declared as shown here:

```
foreach(IGrouping<string, string> sites in webAddrs) {
  Console.WriteLine("Web sites grouped by " + sites.Key);
  foreach(string site in sites)
    Console.WriteLine("  " + site);
  Console.WriteLine();
}
```

Use into to Create a Continuation

When using **select** or **group**, you will sometimes want to generate a temporary result that will be used by a subsequent part of the query to produce the final result. This is called a *query continuation* (or just a *continuation* for short), and it is accomplished through the use of **into** with a **select** or **group** clause. It has the following general form:

into *name query-body*

where *name* is the name of the range variable that iterates over the temporary result and is used by the continuing query, specified by *query-body*. This is why **into** is called a query continuation when used with **select** or **group**—it continues the query. In essence, a query continuation embodies the concept of building a new query that queries the results of the preceding query.

NOTE *There is also a form of* ***into*** *that can be used with* ***join****, which creates a* group join. *This is described later in this chapter.*

Here is an example that uses **into** with **group**. The following program reworks the **GroupDemo** example shown earlier, which creates a list of websites grouped by top-level domain name. In this case, the initial results are queried by a range variable called **ws**. This result is then filtered to remove all groups that have fewer than three elements.

```
// Use into with group.
using System;
using System.Linq;

class IntoDemo {

  static void Main() {

    string[] websites = { "hsNameA.com", "hsNameB.net", "hsNameC.net",
                          "hsNameD.com", "hsNameE.org", "hsNameF.org",
                          "hsNameG.tv",  "hsNameH.net", "hsNameI.tv" };
```

```
    // Create a query that groups websites by top-level domain name,
    // but select only those groups that have more than two members.
    // Here, ws is the range variable over the set of groups
    // returned when the first half of the query is executed.
    var webAddrs = from addr in websites
                   where addr.LastIndexOf(".") != -1
                   group addr by addr.Substring(addr.LastIndexOf("."))
                              into ws
                   where ws.Count() > 2
                   select ws;

    // Execute the query and display the results.
    Console.WriteLine("Top-level domains with more than 2 members.\n");

    foreach(var sites in webAddrs) {
      Console.WriteLine("Contents of " + sites.Key + " domain:");
      foreach(var site in sites)
        Console.WriteLine("  " + site);
      Console.WriteLine();
    }
  }
}
```

The following output is produced:

```
Top-level domains with more than 2 members.

Contents of .net domain:
  hsNameB.net
  hsNameC.net
  hsNameH.net
```

As the output shows, only the **.net** group is returned because it is the only group that has more than two elements.

In the program, pay special attention to this sequence of clauses in the query:

```
group addr by addr.Substring(addr.LastIndexOf(".", addr.Length))
           into ws
where ws.Count() > 2
select ws;
```

First, the results of the **group** clause are stored (creating a temporary result) and a new query begins that operates on the stored results. The range variable of the new query is **ws**. At this point, **ws** will range over each group returned by the first query. (It ranges over groups because the first query results in a sequence of groups.) Next, the **where** clause filters the query so the final result contains only those groups that contain more than two members. This determination is made by calling **Count( )**, which is an *extension method* that is implemented for all **IEnumerable** objects. It returns the number of elements in a sequence. (You'll learn more about extension methods later in this chapter.) The resulting sequence of groups is returned by the **select** clause.

Use let to Create a Variable in a Query

In a query, you will sometimes want to retain a value temporarily. For example, you might want to create an enumerable variable that can, itself, be queried. Or, you might want to store a value that will be used later on in a **where** clause. Whatever the purpose, these types of actions can be accomplished through the use of **let**.

The **let** clause has this general form:

let *name* = *expression*

Here, *name* is an identifier that is assigned the value of *expression*. The type of *name* is inferred from the type of the expression.

Here is an example that shows how **let** can be used to create another enumerable data source. The query takes as input an array of strings. It then converts those strings into **char** arrays. This is accomplished by use of another **string** method called **ToCharArray( )**, which returns an array containing the characters in the string. The result is assigned to a variable called **chrArray**, which is then used by a nested **from** clause to obtain the individual characters in the array. The query then sorts the characters and returns the resulting sequence.

```
// Use a let clause and a nested from clause.
using System;
using System.Linq;

class LetDemo {

  static void Main() {

    string[] strs = { "alpha", "beta", "gamma" };

    // Create a query that obtains the characters in the
    // strings, returned in sorted order. Notice the use
    // of a nested from clause.
    var chrs = from str in strs
               let chrArray = str.ToCharArray()
                  from ch in chrArray
                  orderby ch
                  select ch;

    Console.WriteLine("The individual characters in sorted order:");

    // Execute the query and display the results.
    foreach(char c in chrs) Console.Write(c + " ");

    Console.WriteLine();
  }
}
```

The output is shown here:

```
The individual characters in sorted order:
a a a a a b e g h l m m p t
```

In the program, notice how the **let** clause assigns to **chrArray** a reference to the array returned by **str.ToCharArray( )**:

```
let chrArray = str.ToCharArray()
```

After the **let** clause, other clauses can make use of **chrArray**. Furthermore, because all arrays in C# are convertible to **IEnumerable<T>**, **chrArray** can be used as a data source for a second, nested **from** clause. This is what happens in the example. It uses the nested **from** to enumerate the individual characters in the array, sorting them into ascending sequence and returning the result.

You can also use a **let** clause to hold a non-enumerable value. For example, the following is a more efficient way to write the query used in the **IntoDemo** program shown in the preceding section.

```
var webAddrs = from addr in websites
               let idx = addr.LastIndexOf(".")
               where idx != -1
               group addr by addr.Substring(idx)
                          into ws
               where ws.Count() > 2
               select ws;
```

In this version, the index of the last occurrence of a period is assigned to **idx**. This value is then used by **Substring( )**. This prevents the search for the period from having to be conducted twice.

Join Two Sequences with join

When working with databases, it is common to want to create a sequence that correlates data from two different data sources. For example, an online store might have one database that associates the name of an item with its item number, and a second database that associates the item number with its in-stock status. Given this situation, you might want to generate a list that shows the in-stock status of items by name, rather than by item number. You can do this by correlating the data in the two databases. Such an action is easy to accomplish in LINQ through the use of the **join** clause.

The general form of **join** is shown here (in context with the **from**):

from *range-varA* in *data-sourceA*
  join *range-varB* in *data-sourceB*
    on *range-varA.property* equals *range-varB.property*

The key to using **join** is to understand that each data source must contain data in common, and that data can be compared for equality. Thus, in the general form, *data-sourceA* and *data-sourceB* must have something in common that can be compared. The items being compared are specified by the **on** section. Thus, when *range-varA.property* is equal to *range-varB.property*, the correlation succeeds. In essence, **join** acts like a filter, allowing only those elements that share a common value to pass through.

When using **join**, often the sequence returned is a composite of portions of the two data sources. Therefore, **join** lets you generate a new list that contains elements from two different data sources. This enables you to organize data in a new way.

The following program creates a class called **Item**, which encapsulates an item's name with its number. It creates another class called **InStockStatus**, which links an item number with a Boolean property that indicates whether or not the item is in stock. It also creates a class called **Temp**, which has two fields: one **string** and one **bool**. Objects of this class will hold the result of the query. The query uses **join** to produce a list in which an item's name is associated with its in-stock status.

```
// Demonstrate join.
using System;
using System.Linq;

// A class that links an item name with its number.
class Item {
  public string Name { get; set; }
  public int ItemNumber { get; set; }

  public Item(string n, int inum) {
    Name = n;
    ItemNumber = inum;
  }
}

// A class that links an item number with its in-stock status.
class InStockStatus {
  public int ItemNumber { get; set; }
  public bool InStock { get; set; }

  public InStockStatus(int n, bool b) {
    ItemNumber = n;
    InStock = b;
  }
}

// A class that encapsulates a name with its status.
class Temp {
  public string Name { get; set; }
  public bool InStock { get; set; }

  public Temp(string n, bool b) {
    Name = n;
    InStock  = b;
  }
}

class JoinDemo {
  static void Main() {

    Item[] items = {
         new Item("Pliers", 1424),
         new Item("Hammer", 7892),
         new Item("Wrench", 8534),
         new Item("Saw", 6411)
    };
```

```
    InStockStatus[] statusList = {
         new InStockStatus(1424, true),
         new InStockStatus(7892, false),
         new InStockStatus(8534, true),
         new InStockStatus(6411, true)
    };


    // Create a query that joins Item with InStockStatus to
    // produce a list of item names and availability. Notice
    // that a sequence of Temp objects is produced.
    var inStockList = from item in items
                      join entry in statusList
                        on item.ItemNumber equals entry.ItemNumber
                      select new Temp(item.Name, entry.InStock);

    Console.WriteLine("Item\tAvailable\n");

    // Execute the query and display the results.
    foreach(Temp t in inStockList)
      Console.WriteLine("{0}\t{1}", t.Name, t.InStock);
  }
}
```

The output is shown here:

```
Item    Available

Pliers  True
Hammer  False
Wrench  True
Saw     True
```

To understand how **join** works, let's walk through each line in the query. The query begins in the normal fashion with this **from** clause:

```
var inStockList = from item in items
```

This clause specifies that **item** is the range variable for the data source specified by **items**. The **items** array contains objects of type **Item**, which encapsulate a name and a number for an inventory item.

Next comes the **join** clause shown here:

```
join entry in statusList
  on item.ItemNumber equals entry.ItemNumber
```

This clause specifies that **entry** is the range variable for the **statusList** data source. The **statusList** array contains objects of type **InStockStatus**, which link an item number with its status. Thus, **items** and **statusList** have a property in common: the item number. This is used by the **on**/**equals** portion of the **join** clause to describe the correlation. Thus, **join** matches items from the two data sources when their item numbers are equal.

Finally, the **select** clause returns a **Temp** object that contains an item's name along with its in-stock status:

```
select new Temp(item.Name, entry.InStock);
```

Therefore, the sequence obtained by the query consists of **Temp** objects.

Although the preceding example is fairly straightforward, **join** supports substantially more sophisticated operations. For example, you can use **into** with **join** to create a *group join*, which creates a result that consists of an element from the first sequence and a group of all matching elements from the second sequence. (You'll see an example of this a bit later in this chapter.) In general, the time and effort needed to fully master **join** is well worth the investment because it gives you the ability to reorganize data at runtime. This is a powerful capability. This capability is made even more powerful by the use of anonymous types, described in the next section.

Anonymous Types

C# 3.0 adds a new feature called the *anonymous type* that directly relates to LINQ. As the name implies, an anonymous type is a class that has no name. Its primary use is to create an object returned by the **select** clause. Often, the outcome of a query is a sequence of objects that are either a composite of two (or more) data sources (such as in the case of **join**) or include a subset of the members of one data source. In either case, often the type of the object being returned is needed only because of the query and is not used elsewhere in the program. In this case, using an anonymous type eliminates the need to declare a class that will be used simply to hold the outcome of the query.

An anonymous type is created through the use of this general form:

new { *nameA* = *valueA*, *nameB* = *valueB*, ... }

Here, the names specify identifiers that translate into read-only properties that are initialized by the values. For example,

```
new { Count = 10, Max = 100, Min = 0 }
```

This creates a class type that has three public read-only properties: **Count**, **Max**, and **Min**. These are given the values 10, 100, and 0, respectively. These properties can be referred to by name by other code. Notice that an anonymous type uses object initializers to initialize fields and properties. As explained in Chapter 8, object initializers provide a way to initialize an object without explicitly invoking a constructor. This is necessary in the case of anonymous types because there is no way to explicitly call a constructor. (Recall that constructors have the same name as their class. In the case of an anonymous class, there is no name. So, how would you invoke the constructor?)

Because an anonymous type has no name, you must use an implicitly typed variable to refer to it. This lets the compiler infer the proper type. For example,

```
var myOb = new { Count = 10, Max = 100, Min = 0 }
```

creates a variable called **myOb** that is assigned a reference to the object created by the anonymous type expression. This means that the following statements are legal:

```
Console.WriteLine("Count is " + myOb.Count);

if(i <= myOb.Max && i >= myOb.Min) // ...
```

Remember, when an anonymous type is created, the identifiers that you specify become read-only public properties. Thus, they can be used by other parts of your code.

Although the term *anonymous type* is used, it's not quite completely true! The type is anonymous relative to you, the programmer. However, the compiler does give it an internal name. Thus, anonymous types do not violate C#'s strong type checking rules.

To fully understand the value of anonymous types, consider this rewrite of the previous program that demonstrated **join**. Recall that in the previous version, a class called **Temp** was needed to encapsulate the result of the **join**. Through the use of an anonymous type, this "placeholder" class is no longer needed and no longer clutters the source code to the program. The output from the program is unchanged from before.

```
// Use an anonymous type to improve the join demo program.
using System;
using System.Linq;

// A class that links an item name with its number.
class Item {
  public string Name { get; set; }
  public int ItemNumber { get; set; }

  public Item(string n, int inum) {
    Name = n;
    ItemNumber = inum;
  }
}

// A class that links an item number with its in-stock status.
class InStockStatus {
  public int ItemNumber { get; set; }
  public bool InStock { get; set; }

  public InStockStatus(int n, bool b) {
    ItemNumber = n;
    InStock = b;
  }
}

class AnonTypeDemo {
  static void Main() {

    Item[] items = {
         new Item("Pliers", 1424),
         new Item("Hammer", 7892),
         new Item("Wrench", 8534),
         new Item("Saw", 6411)
    };

    InStockStatus[] statusList = {
         new InStockStatus(1424, true),
         new InStockStatus(7892, false),
         new InStockStatus(8534, true),
         new InStockStatus(6411, true)
    };
```

```
    // Create a query that joins Item with InStockStatus to
    // produce a list of item names and availability.
    // Now, an anonymous type is used.
    var inStockList = from item in items
                      join entry in statusList
                        on item.ItemNumber equals entry.ItemNumber
                      select new { Name = item.Name,
                                   InStock =  entry.InStock };

    Console.WriteLine("Item\tAvailable\n");

    // Execute the query and display the results.
    foreach(var t in inStockList)
      Console.WriteLine("{0}\t{1}", t.Name, t.InStock);
  }
}
```

Pay special attention to the **select** clause:

```
select new { Name = item.Name,
             InStock =  entry.InStock };
```

It returns an object of an anonymous type that has two read-only properties, **Name** and **InStock**. These are given the values specified by the item's name and availability. Because of the anonymous type, there is no longer any need for the **Temp** class.

One other point. Notice the **foreach** loop that executes the query. It now uses **var** to declare the iteration variable. This is necessary because the type of the object contained in **inStockList** has no name. This situation is one of the reasons that C# 3.0 added implicitly typed variables. They are needed to support anonymous types.

Before moving on, there is one more aspect of anonymous types that warrants a mention. In some cases, including the one just shown, you can simplify the syntax of the anonymous type through the use of a *projection initializer.* In this case, you simply specify the name of the initializer by itself. This name automatically becomes the name of the property. For example, here is another way to code the **select** clause used by the preceding program:

```
select new { item.Name, entry.InStock };
```

Here, the property names are still **Name** and **InStock**, just as before. The compiler automatically "projects" the identifiers **Name** and **InStock**, making them the property names of the anonymous type. Also as before, the properties are given the values specified by **item.Name** and **entry.InStock**.

Create a Group Join

As explained earlier, you can use **into** with **join** to create a *group join,* which creates a sequence in which each entry in the result consists of an entry from the first sequence and a group of all matching elements from the second sequence. No example was presented then because often a group join makes use of an anonymous type. Now that anonymous types have been covered, an example of a simple group join can be given.

The following example uses a group join to create a list in which various transports, such as cars, boats, and planes, are organized by their general transportation category, which is

land, sea, and air. The program first creates a class called **Transport** that links a transport type with its classification. Inside **Main( )**, it creates two input sequences. The first is an array of strings that contains the names of the general means by which one travels, which is land, sea, and air. The second is an array of **Transport**, which encapsulates various means of transportation. It then uses a group join to produce a list of transports that are organized by their category.

```
// Demonstrate a simple group join.
using System;
using System.Linq;

// This class links the name of a transport, such as Train,
// with its general classification, such as land, sea, or air.
class Transport {
  public string Name { get; set; }
  public string How { get; set; }

  public Transport(string n, string h) {
    Name = n;
    How = h;
  }
}

class GroupJoinDemo {
  static void Main() {

    // An array of transport classifications.
    string[] travelTypes = {
         "Air",
         "Sea",
         "Land"
    };

    // An array of transports.
    Transport[] transports = {
         new Transport("Bicycle", "Land"),
         new Transport("Balloon", "Air"),
         new Transport("Boat", "Sea"),
         new Transport("Jet", "Air"),
         new Transport("Canoe", "Sea"),
         new Transport("Biplane", "Air"),
         new Transport("Car", "Land"),
         new Transport("Cargo Ship", "Sea"),
         new Transport("Train", "Land")
    };

    // Create a query that uses a group join to produce
    // a list of item names and IDs organized by category.
    var byHow = from how in travelTypes
                      join trans in transports
                      on how equals trans.How
                      into lst
                    select new { How = how, Tlist = lst };
```

```
    // Execute the query and display the results.
    foreach(var t in byHow) {
      Console.WriteLine("{0} transportation includes:", t.How);

      foreach(var m in t.Tlist)
        Console.WriteLine("  " + m.Name);

      Console.WriteLine();
    }
  }
}
```

The output is shown here:

```
Air transportation includes:
  Balloon
  Jet
  Biplane

Sea transportation includes:
  Boat
  Canoe
  Cargo Ship

Land transportation includes:
  Bicycle
  Car
  Train
```

The key part of the program is, of course, the query, which is shown here:

```
var byHow = from how in travelTypes
                  join trans in transports
                  on how equals trans.How
                  into lst
                select new { How = how, Tlist = lst };
```

Here is how it works. The **from** statement uses **how** to range over the **travelTypes** array. Recall that **travelTypes** contains an array of the general travel classifications: air, land, and sea. The **join** clause joins each travel type with those transports that use that type. For example, the type Land is joined with Bicycle, Car, and Train. However, because of the **into** clause, for each travel type, the **join** produces a list of the transports that use that travel type. This list is represented by **lst**. Finally, **select** returns an anonymous type that encapsulates each value of **how** (the travel type) with a list of transports. This is why the two **foreach** loops shown here are needed to display the results of the query:

```
foreach(var t in byHow) {
  Console.WriteLine("{0} transportation includes:", t.How);

  foreach(var m in t.Tlist)
    Console.WriteLine("  " + m.Name);

  Console.WriteLine();
}
```

The outer loop obtains an object that contains the name of the travel type and the list of the transports for that type. The inner loop displays the individual transports.

The Query Methods

The query syntax described by the preceding sections is the way you will probably write most queries in C#. It is convenient, powerful, and compact. It is, however, not the only way to write a query. The other way is to use the *query methods*. These methods can be called on any enumerable object, such as an array.

The Basic Query Methods

The query methods are defined by **System.Linq.Enumerable** and are implemented as *extension methods* that extend the functionality of **IEnumerable<T>**. (Query methods are also defined by **System.Linq.Queryable**, which extends the functionality of **IQueryable<T>**, but this interface is not used in this chapter.) An extension method adds functionality to another class, but without the use of inheritance. Support for extension methods was added by C# 3.0, and we will look more closely at them later in this chapter. For now, it is sufficient to understand that query methods can be called only on an object that implements **IEnumerable<T>**.

The **Enumerable** class provides many query methods, but at the core are those that correspond to the query keywords described earlier. These methods are shown here, along with the keywords to which they relate. Understand that these methods have overloaded forms and only their simplest form is shown. However, this is also the form that you will often use.

Query Keyword	Equivalent Query Method
select	Select(*arg*)
where	Where(*arg*)
orderby	OrderBy(*arg*) or OrderByDescending(*arg*)
join	Join(*seq2*, *key1*, *key2*, *result*)
group	GroupBy(*arg*)

Except for **Join( )**, the other methods take one argument, *arg*, which is an object of type **Func<T, TResult>**, as a parameter. This is a delegate type defined by LINQ. It is declared like this:

```
delegate TResult Func<T, TResult>(T arg)
```

Here, **TResult** specifies the result of the delegate and **T** specifies the parameter type. In the query methods, *arg* determines what action the query method takes. For example, in the case of **Where( )**, *arg* determines how the query filters the data. Each of these query methods returns an enumerable object. Thus, the result of one can be used to execute a call on another, allowing the methods to be chained together.

The **Join( )** method takes four arguments. The first is a reference to the second sequence to be joined. The first sequence is the one on which **Join( )** is called. The key selector for the first sequence is passed via *key1*, and the key selector for the second sequence is passed via

key2. The result of the join is described by *result*. The type of *key1* is **Func<TOuter, TKey>**, and the type of *key2* is **Func<TInner, TKey>**. The *result* argument is of type **Func<TOuter, TInner, TResult>**. Here, **TOuter** is the element type of the invoking sequence; **TInner** is the element type of the passed sequence; and **TResult** is the type of the resulting elements. An enumerable object is returned that contains the result of the join.

Although an argument to a query method such as **Where( )** is a method compatible with the specified form of the **Func** delegate, it does not need to be an explicitly declared method. In fact, most often it won't be. Instead, you will usually use a lambda expression. As explained in Chapter 15, a lambda expression is a new syntactic feature provided by C# 3.0. It offers a streamlined, yet powerful way to define what is, essentially, an anonymous method. The C# compiler automatically converts a lambda expression into a form that can be passed to a **Func** parameter. Because of the streamlined convenience offered by lambda expressions, they are used by all of the examples in this section.

Create Queries by Using the Query Methods

By using the query methods in conjunction with lambda expressions, it is possible to create queries that do not use the C# query syntax. Instead, the query methods are called. Let's begin with a simple example. It reworks the first program in this chapter so that it uses calls to **Where( )** and **Select( )** rather than the query keywords.

```
// Use the query methods to create a simple query.
// This is a reworked version of the first program in this chapter.
using System;
using System.Linq;

class SimpQuery {
  static void Main() {

    int[] nums =  { 1, -2, 3, 0, -4, 5 };

    // Use Where() and Select() to create a simple query.
    var posNums = nums.Where(n => n > 0).Select(r => r);

    Console.Write("The positive values in nums: ");

    // Execute the query and display the results.
    foreach(int i in posNums) Console.Write(i + " ");
    Console.WriteLine();
  }
}
```

The output, shown here, is the same as the original version:

```
The positive values in nums: 1 3 5
```

In the program, pay special attention to this line:

```
var posNums = nums.Where(n => n > 0).Select(r => r);
```

This creates a query called **posNums** that creates a sequence of the positive values in **nums**. It does this by use of the **Where( )** method (to filter the values) and **Select( )** (to select the

values). The **Where( )** method can be invoked on **nums** because all arrays implement **IEnumerable<T>**, which supports the query extension methods.

Technically, the **Select( )** method in the preceding example is not necessary because in this simple case, the sequence returned by **Where( )** already contains the result. However, you can use more sophisticated selection criteria, just as you did with the query syntax. For example, this query returns the positive values in **nums** increased by an order of magnitude:

```
var posNums = nums.Where(n => n > 0).Select(r => r * 10);
```

As you might expect, you can chain together other operations. For example, this query selects the positive values, sorts them into descending order, and returns the resulting sequence:

```
var posNums = nums.Where(n => n > 0).OrderByDescending(j => j);
```

Here, the expression **j => j** specifies that the ordering is dependent on the input parameter, which is an element from the sequence obtained from **Where( )**.

Here is an example that demonstrates the **GroupBy( )** method. It reworks the **group** example shown earlier.

```
// Demonstrate the GroupBy() query method.
// This program reworks the earlier version that used
// the query syntax.
using System;
using System.Linq;

class GroupByDemo {

  static void Main() {

    string[] websites = { "hsNameA.com", "hsNameB.net", "hsNameC.net",
                          "hsNameD.com", "hsNameE.org", "hsNameF.org",
                          "hsNameG.tv",  "hsNameH.net", "hsNameI.tv" };

    // Use query methods to group websites by top-level domain name.
    var webAddrs = websites.Where(w => w.LastIndexOf(".") != 1).
           GroupBy(x => x.Substring(x.LastIndexOf(".", x.Length)));

    // Execute the query and display the results.
    foreach(var sites in webAddrs) {
      Console.WriteLine("Web sites grouped by " + sites.Key);
      foreach(var site in sites)
        Console.WriteLine("  " + site);
      Console.WriteLine();
    }
  }
}
```

This version produces the same output as the earlier version. The only difference is how the query is created. In this version, the query methods are used.

Here is another example. Recall the **join** query used in the **JoinDemo** example shown earlier:

```
var inStockList = from item in items
                  join entry in statusList
                    on item.ItemNumber equals entry.ItemNumber
                  select new Temp(item.Name, entry.InStock);
```

This query produces a sequence that contains objects that encapsulate the name and the in-stock status of an inventory item. This information is synthesized from joining the two lists **items** and **statusList**. The following version reworks this query so that it uses the **Join( )** method rather than the C# query syntax:

```
// Use Join() to produce a list of item names and status.
var inStockList = items.Join(statusList,
                    k1 => k1.ItemNumber,
                    k2 => k2.ItemNumber,
                    (k1, k2) => new Temp(k1.Name, k2.InStock) );
```

Although this version uses the named class called **Temp** to hold the resulting object, an anonymous type could have been used instead. This approach is shown next:

```
var inStockList = items.Join(statusList,
                    k1 => k1.ItemNumber,
                    k2 => k2.ItemNumber,
                    (k1, k2) => new { k1.Name, k2.InStock} );
```

Query Syntax vs. Query Methods

As the preceding section has explained, C# has two ways of creating queries: the query syntax and the query methods. What is interesting, and not readily apparent by simply looking at a program's source code, is that the two approaches are more closely related than you might at first assume. The reason is that the query syntax is compiled into calls to the query methods. Thus, when you write something like

```
where x < 10
```

the compiler translates it into

```
Where(x => x < 10)
```

Therefore, the two approaches to creating a query ultimately lead to the same place.

Given that the two approaches are ultimately equivalent, the following question naturally arises: Which approach is best for a C# program? The answer: In general, you will want to use the query syntax. It is fully integrated into the C# language, supported by keywords and syntax, and is cleaner.

More Query-Related Extension Methods

In addition to the methods that correspond to the query keywords supported by C#, the .NET Framework provides several other query-related extension methods that are often helpful in a query. These query-related methods are defined for **IEnumerable<T>** by **Enumerable**. Here is a sampling of several commonly used methods. Because many of the methods are overloaded, only their general form is shown.

Method	Description
All(*condition*)	Returns true if all elements in a sequence satisfy a specified condition.
Any(*condition*)	Returns true if any element in a sequence satisfies a specified condition.
Average( )	Returns the average of the values in a numeric sequence.
Contains(*obj*)	Returns true if the sequence contains the specified object.
Count( )	Returns the length of a sequence. This is the number of elements that it contains.
First( )	Returns the first element in a sequence.
Last( )	Returns the last element in a sequence.
Max( )	Returns the maximum value in a sequence.
Min( )	Returns the minimum value in a sequence.
Sum( )	Returns the summation of the values in a numeric sequence.

You have already seen **Count( )** in action earlier in this chapter. Here is a program that demonstrates the others:

```
// Use several of the extension methods defined by Enumerable.
using System;
using System.Linq;

class ExtMethods {
  static void Main() {

    int[] nums =  { 3, 1, 2, 5, 4 };

    Console.WriteLine("The minimum value is " + nums.Min());
    Console.WriteLine("The maximum value is " + nums.Max());

    Console.WriteLine("The first value is " + nums.First());
    Console.WriteLine("The last value is " + nums.Last());

    Console.WriteLine("The sum is " + nums.Sum());
    Console.WriteLine("The average is " + nums.Average());

    if(nums.All(n => n > 0))
      Console.WriteLine("All values are greater than zero.");

    if(nums.Any(n => (n % 2) == 0))
      Console.WriteLine("At least one value is even.");

    if(nums.Contains(3))
      Console.WriteLine("The array contains 3.");
  }
}
```

The output is shown here:

```
The minimum value is 1
The maximum value is 5
The first value is 3
The last value is 4
The sum is 15
The average is 3
All values are greater than zero.
At least one value is even.
The array contains 3.
```

You can also use the query-related extension methods within a query based on the C# query syntax. In fact, it is quite common to do so. For example, this program uses **Average( )** to obtain a sequence that contains only those values that are less than the average of the values in an array.

```
// Use Average() with the query syntax.
using System;
using System.Linq;

class ExtMethods2 {
  static void Main() {

    int[] nums =  { 1, 2, 4, 8, 6, 9, 10, 3, 6, 7 };

    var ltAvg = from n in nums
                let x = nums.Average()
                where n < x
                select n;

    Console.WriteLine("The average is " + nums.Average());

    Console.Write("These values are less than the average: ");

    // Execute the query and display the results.
    foreach(int i in ltAvg) Console.Write(i + " ");

    Console.WriteLine();
  }
}
```

The output is shown here:

```
The average is 5.6
These values are less than the average: 1 2 4 3
```

Pay special attention to the query:

```
    var ltAvg = from n in nums
                let x = nums.Average()
                where n < x
                select n;
```

Notice in the **let** statement, **x** is set equal to the average of the values in **nums**. This value is obtained by calling **Average( )** on **nums**.

Deferred vs. Immediate Query Execution

In LINQ, queries have two different modes of execution: immediate and deferred. As explained early in this chapter, a query defines a set of rules that are not actually executed until a **foreach** statement executes. This is called *deferred execution.*

However, if you use one of the extension methods that produces a nonsequence result, then the query must be executed to obtain that result. For example, consider the **Count( )** method. In order for **Count( )** to return the number of elements in the sequence, the query must be executed, and this is done automatically when **Count( )** is called. In this case, *immediate execution* takes place, with the query being executed automatically in order to obtain the result. Therefore, even though you don't explicitly use the query in a **foreach** loop, the query is still executed.

Here is a simple example. It obtains the number of positive elements in the sequence.

```
// Use immediate execution.
using System;
using System.Linq;

class ImmediateExec {
  static void Main() {

    int[] nums =  { 1, -2, 3, 0, -4, 5 };

    // Create a query that obtains the number of positive
    // values in nums.
    int len = (from n in nums
               where n > 0
               select n).Count();

    Console.WriteLine("The number of positive values in nums: " + len);
  }
}
```

The output is

```
The number of positive values in nums: 3
```

In the program, notice that no explicit **foreach** loop is specified. Instead, the query automatically executes because of the call to **Count( )**.

As a point of interest, the query in the preceding program could also have been written like this:

```
var posNums = from n in nums
              where n > 0
              select n;

int len = posNums.Count(); // query executes here
```

In this case, **Count( )** is called on the query variable. At that point, the query is executed to obtain the count.

Two other methods that cause immediate execution of a query are **ToArray( )** and **ToList( )**. Both are extension methods defined by **Enumerable**. **ToArray( )** returns the results

of a query in an array. **ToList( )** returns the results of a query in the form of a **List** collection. (See Chapter 24 for a discussion of collections.) In both cases, the query is executed to obtain the results. For example, the following sequence obtains an array of the results generated by the **posNums** query just shown. It then displays the results.

```
int[] pnums = posNum.ToArray(); // query executes here

foreach(int i in pnums)
  Console.Write(i + " ");
}
```

Expression Trees

Another new LINQ-related feature is the *expression tree*. An expression tree is a representation of a lambda expression as data. Thus, an expression tree, itself, cannot be executed. It can, however, be converted into an executable form. Expression trees are encapsulated by the **System.Linq.Expressions.Expression<T>** class. Expression trees are useful in situations in which a query will be executed by something outside the program, such as a database that uses SQL. By representing the query as data, the query can be converted into a format understood by the database. This process is used by the LINQ to SQL feature provided by Visual C#, for example. Thus, expression trees help C# support a variety of data sources.

You can obtain an executable form of an expression tree by calling the **Compile( )** method defined by **Expression**. It returns a reference that can be assigned to a delegate and then executed. You can declare your own delegate type or use one of the predefined **Func** delegate types defined within the **System** namespace. Two forms of the **Func** delegate were mentioned earlier, when the query methods were described. Here is a list of all its forms:

delegate TResult Func<TResult>( )

delegate TResult Func<T1, TResult>( )

delegate TResult Func<T1, T2, TResult>( )

delegate TResult Func<T1, T2, T3, TResult>( )

delegate TResult Func<T1, T2, T3, T4, TResult>( )

These forms represent methods that return a value and take from zero to four parameters (whose types are **T1** through **T4**). If your expression requires more than four parameters, then you will need to define your own delegate type.

Expression trees have one key restriction: Only expression lambdas can be represented by expression trees. They cannot be used to represent statement lambdas.

Here is a simple example of an expression tree in action. It creates an expression tree whose data represents a method that determines if one integer is a factor of another. It then compiles the expression tree into executable code. Finally, it demonstrates the compiled code.

```
// A simple expression tree.
using System;
using System.Linq;
using System.Linq.Expressions;
```

```
class SimpleExpTree {
  static void Main() {

    // Represent a lambda expression as data.
    Expression<Func<int, int, bool>>
      IsFactorExp = (n, d) => (d != 0) ? (n % d) == 0 : false;

    // Compile the expression data into executable code.
    Func<int, int, bool> IsFactor = IsFactorExp.Compile();

    // Execute the expression.
    if(IsFactor(10, 5))
      Console.WriteLine("5 is a factor of 10.");

    if(!IsFactor(10, 7))
      Console.WriteLine("7 is not a factor of 10.");

    Console.WriteLine();
  }
}
```

The output is shown here:

```
5 is a factor of 10.
7 is not a factor of 10.
```

The program illustrates the two key steps in using an expression tree. First, it creates an expression tree by using this statement:

```
Expression<Func<int, int, bool>>
  IsFactorExp = (n, d) => (d != 0) ? (n % d) == 0 : false;
```

This constructs a representation of a lambda expression in memory. As explained, this representation is data, not code. This representation is referred to by **IsFactorExp**. The following statement converts the expression data into executable code:

```
Func<int, int, bool> IsFactor = IsFactorExp.Compile();
```

After this statement executes, the **IsFactor** delegate can be called to determine if one value is a factor of another.

One other point: Notice that **Func<int, int, bool>** indicates the delegate type. This form of **Func** specifies two parameters of type **int** and a return type of **bool**. This is the form of **Func** that is compatible with the lambda expression used in the program because that expression requires two parameters. Other lambda expressions may require different forms of **Func**, based on the number of parameters they require. In general, the specific form of **Func** must match the requirements of the lambda expression.

Extension Methods

As mentioned earlier, extension methods provide a means by which functionality can be added to a class without using the normal inheritance mechanism. Although you won't often create your own extension methods (because the inheritance mechanism offers a

better solution in many cases), it is still important that you understand how they work because of their integral importance to LINQ.

An extension method is a static method that must be contained within a static, non-generic class. The type of its first parameter determines the type of objects on which the extension method can be called. Furthermore, the first parameter must be modified by **this**. The object on which the method is invoked is passed automatically to the first parameter. It is not explicitly passed in the argument list. A key point is that even though an extension method is declared **static**, it can still be called on an object, just as if it were an instance method.

Here is the general form of an extension method:

static *ret-type name*(this *invoked-on-type ob*, *param-list*)

Of course, if there are no arguments other than the one passed implicitly to *ob,* then *param-list* will be empty. Remember, the first parameter is automatically passed the object on which the method is invoked. In general, an extension method will be a public member of its class.

Here is an example that creates three simple extension methods:

```
// Create and use some extension methods.
using System;

static class MyExtMeths {

  // Return the reciprocal of a double.
  public static double Reciprocal(this double v) {
    return 1.0 / v;
  }

  // Reverse the case of letters within a string and
  // return the result.
  public static string RevCase(this string str) {
    string temp = "";

    foreach(char ch in str) {
      if(Char.IsLower(ch)) temp += Char.ToUpper(ch);
      else temp += Char.ToLower(ch);
    }
    return temp;
  }

  // Return the absolute value of n / d.
  public static double AbsDivideBy(this double n, double d) {
    return Math.Abs(n / d);
  }
}

class ExtDemo {
  static void Main() {
    double val = 8.0;
    string str = "Alpha Beta Gamma";

    // Call the Recip() extension method.
    Console.WriteLine("Reciprocal of {0} is {1}",
                      val, val.Reciprocal());
```

```
    // Call the RevCase() extension method.
    Console.WriteLine(str + " after reversing case is " +
                      str.RevCase());

    // Use AbsDivideBy();
    Console.WriteLine("Result of val.AbsDivideBy(-2): " +
                      val.AbsDivideBy(-2));
  }
}
```

The output is shown here:

```
Reciprocal of 8 is 0.125
Alpha Beta Gamma after reversing case is aLPHA bETA gAMMA
Result of val.AbsDivideBy(-2): 4
```

In the program, notice that each extension method is contained in a static class called **MyExtMeths**. As explained, an extension method must be declared within a static class. Furthermore, this class must be in scope in order for the extension methods that it contains to be used. (This is why you needed to include the **System.Linq** namespace to use the LINQ-related extension methods.) Next, notice the calls to the extension methods. They are invoked on an object, in just the same way that an instance method is called. The main difference is that the invoking object is passed to the first parameter of the extension method. Therefore, when the expression

```
val.AbsDivideBy(-2)
```

executes, **val** is passed to the **n** parameter of **AbsDivideBy( )** and –2 is passed to the **d** parameter.

As a point of interest, because the methods **Reciprocal( )** and **AbsDivideBy( )** are defined for **double**, it is legal to invoke them on a **double** literal, as shown here:

```
8.0.Reciprocal()
8.0.AbsDivideBy(-1)
```

Furthermore, **RevCase( )** can be invoked like this:

```
"AbCDe".RevCase()
```

Here, the reversed-case version of a string literal is returned.

20
CHAPTER

Unsafe Code, Pointers, Nullable Types, and Miscellaneous Topics

This chapter covers a feature of C# whose name usually takes programmers by surprise: unsafe code. Unsafe code often involves the use of pointers. Together, unsafe code and pointers enable C# to be used to create applications that one might normally associate with C++: high-performance, systems code. Moreover, the inclusion of unsafe code and pointers gives C# capabilities that are lacking in Java.

Also covered in this chapter are nullable types, partial class and partial method definitions, and fixed-size buffers. The chapter concludes by discussing the few keywords that have not been covered by the preceding chapters.

Unsafe Code

C# allows you to write what is called "unsafe" code. Although this statement might seem shocking, it really isn't. Unsafe code is not code that is poorly written; it is code that does not execute under the full management of the common language runtime (CLR). As explained in Chapter 1, C# is normally used to create managed code. It is possible, however, to write code that does not execute under the full control of the CLR. This unmanaged code is not subject to the same controls and constraints as managed code, so it is called "unsafe" because it is not possible to verify that it won't perform some type of harmful action. Thus, the term *unsafe* does not mean that the code is inherently flawed. It simply means that it is possible for the code to perform actions that are not subject to the supervision of the managed context.

Given that unsafe code might cause problems, you might ask why anyone would want to create such code. The answer is that managed code prevents the use of *pointers*. If you are familiar with C or C++, then you know that pointers are variables that hold the addresses of other objects. Thus, pointers are a bit like references in C#. The main difference is that a pointer can point anywhere in memory; a reference always refers to an object of its type. Because a pointer can point anywhere in memory, it is possible to misuse a pointer. It is also

easy to introduce a coding error when using pointers. This is why C# does not support pointers when creating managed code. Pointers are, however, both useful and necessary for some types of programming (such as system-level utilities), and C# does allow you to create and use pointers. However, all pointer operations must be marked as unsafe since they execute outside the managed context.

The declaration and use of pointers in C# parallels that of C/C++—if you know how to use pointers in C/C++, then you can use them in C#. But remember, the essence of C# is the creation of managed code. Its ability to support unmanaged code allows it to be applied to a special class of problems. It is not for normal C# programming. In fact, to compile unmanaged code, you must use the **/unsafe** compiler option.

Since pointers are at the core of unsafe code, we will begin there.

Pointer Basics

A pointer is a variable that holds the address of some other object, such as another variable. For example, if **x** contains the address of **y**, then **x** is said to "point to" **y**. When a pointer points to a variable, the value of that variable can be obtained or changed through the pointer. Operations through pointers are often referred to as *indirection.*

Declaring a Pointer

Pointer variables must be declared as such. The general form of a pointer variable declaration is

type var-name*;

Here, *type* is the pointer's *referent type,* which must be a nonreference type. Thus, you cannot declare a pointer to a class object. A pointer's referent type is also sometimes called its *base type.* Notice the placement of the *. It follows the type name. *var-name* is the name of the pointer variable.

Here is an example. To declare **ip** to be a pointer to an **int**, use this declaration:

```
int* ip;
```

For a **float** pointer, use

```
float* fp;
```

In general, in a declaration statement, following a type name with an * creates a pointer type.

The type of data that a pointer will point to is determined by its referent type. Thus, in the preceding examples, **ip** can be used to point to an **int**, and **fp** can be used to point to a **float**. Understand, however, that there is nothing that actually prevents a pointer from pointing elsewhere. This is why pointers are potentially unsafe.

If you come from a C/C++ background, then you need to be aware of an important difference between the way C# and C/C++ declare pointers. When you declare a pointer type in C/C++, the * is not distributive over a list of variables in a declaration. Thus, in C/C++, this statement

```
int* p, q;
```

declares an **int** pointer called **p** and an **int** called **q**. It is equivalent to the following two declarations:

```
int* p;
int q;
```

However, in C#, the * *is* distributive and the declaration

```
int* p, q;
```

creates two pointer variables. Thus, in C# it is the same as these two declarations:

```
int* p;
int* q;
```

This is an important difference to keep in mind when porting C/C++ code to C#.

The * and & Pointer Operators

Two operators are used with pointers: * and **&**. The **&** is a unary operator that returns the memory address of its operand. (Recall that a unary operator requires only one operand.) For example,

```
int* ip;
int num = 10;

ip = &num;
```

puts into **ip** the memory address of the variable **num**. This address is the location of the variable in the computer's internal memory. It has *nothing* to do with the *value* of **num**. Thus, **ip** *does not* contain the value 10 (**num**'s initial value). It contains the address at which **num** is stored. The operation of **&** can be remembered as returning "the address of" the variable it precedes. Therefore, the preceding assignment statement could be verbalized as "**ip** receives the address of **num**."

The second operator is *, and it is the complement of **&**. It is a unary operator that evaluates to the value of the variable located at the address specified by its operand. That is, it refers to the value of the variable pointed to by a pointer. Continuing with the same example, if **ip** contains the memory address of the variable **num**, then

```
int val = *ip;
```

will place into **val** the value 10, which is the value of **num**, which is pointed to by **ip**. The operation of * can be remembered as "at address." In this case, then, the statement could be read as "**val** receives the value at address **ip**."

The * can also be used on the left side of an assignment statement. In this usage, it sets the value pointed to by the pointer. For example,

```
*ip = 100;
```

This statement assigns 100 to the variable pointed to by **ip**, which is **num** in this case. Thus, this statement can be read as "at address **ip**, put the value 100."

Using unsafe

Any code that uses pointers must be marked as unsafe by using the **unsafe** keyword. You can mark types (such as classes and structures), members (such as methods and operators),

or individual blocks of code as unsafe. For example, here is a program that uses pointers inside **Main( )**, which is marked unsafe:

```
// Demonstrate pointers and unsafe.

using System;

class UnsafeCode {
  // Mark Main as unsafe.
  unsafe static void Main() {
    int count = 99;
    int* p; // create an int pointer

    p = &count; // put address of count into p

    Console.WriteLine("Initial value of count is " + *p);

    *p = 10; // assign 10 to count via p

    Console.WriteLine("New value of count is " + *p);
  }
}
```

The output of this program is shown here:

```
Initial value of count is 99
New value of count is 10
```

Using fixed

The **fixed** modifier is often used when working with pointers. It prevents a managed variable from being moved by the garbage collector. This is needed when a pointer refers to a field in a class object, for example. Because the pointer has no knowledge of the actions of the garbage collector, if the object is moved, the pointer will point to the wrong object. Here is the general form of **fixed**:

```
fixed (type* p = &fixedObj) {
  // use fixed object
}
```

Here, *p* is a pointer that is being assigned the address of an object. The object will remain at its current memory location until the block of code has executed. You can also use a single statement for the target of a **fixed** statement. The **fixed** keyword can be used only in an unsafe context. You can declare more than one fixed pointer at a time using a comma-separated list.

Here is an example of **fixed**:

```
// Demonstrate fixed.

using System;

class Test {
  public int num;
  public Test(int i) { num = i; }
}
```

```
class FixedCode {
  // Mark Main as unsafe.
  unsafe static void Main() {
    Test o = new Test(19);

    fixed (int* p = &o.num) { // use fixed to put address of o.num into p

      Console.WriteLine("Initial value of o.num is " + *p);

      *p = 10; // assign 10 to count via p

      Console.WriteLine("New value of o.num is " + *p);
    }
  }
}
```

The output from this program is shown here:

```
Initial value of o.num is 19
New value of o.num is 10
```

Here, **fixed** prevents **o** from being moved. Because **p** points to **o.num**, if **o** were moved, then **p** would point to an invalid location.

Accessing Structure Members Through a Pointer

A pointer can point to an object of a structure type as long as the structure does not contain reference types. When you access a member of a structure through a pointer, you must use the arrow operator, which is **–>**, rather than the dot (**.**) operator. For example, given this structure,

```
struct MyStruct {
  public int a;
  public int b;
  public int Sum() { return a + b; }
}
```

you would access its members through a pointer, like this:

```
MyStruct o = new MyStruct();
MyStruct* p; // declare a pointer

p = &o;
p->a = 10; // use the -> operator
p->b = 20; // use the -> operator

Console.WriteLine("Sum is " + p->Sum());
```

Pointer Arithmetic

There are only four arithmetic operators that can be used on pointers: **++**, **– –**, **+**, and **–**. To understand what occurs in pointer arithmetic, we will begin with an example. Let **p1** be an **int** pointer with a current value of 2,000 (that is, it contains the address 2,000). After this expression,

```
p1++;
```

the contents of **p1** will be 2,004, not 2,001! The reason is that each time **p1** is incremented, it will point to the *next* **int**. Since **int** in C# is 4 bytes long, incrementing **p1** increases its value by 4. The reverse is true of decrements. Each decrement decreases **p1**'s value by 4. For example,

```
p1--;
```

will cause **p1** to have the value 1,996, assuming it previously was 2,000.

Generalizing from the preceding example, each time that a pointer is incremented, it will point to the memory location of the next element of its referent type. Each time it is decremented, it will point to the location of the previous element of its referent type.

Pointer arithmetic is not limited to only increment and decrement operations. You can also add or subtract integers to or from pointers. The expression

```
p1 = p1 + 9;
```

makes **p1** point to the ninth element of **p1**'s referent type, beyond the one it is currently pointing to.

Although you cannot add pointers, you can subtract one pointer from another (provided they are both of the same referent type). The remainder will be the number of elements of the referent type that separate the two pointers.

Other than addition and subtraction of a pointer and an integer, or the subtraction of two pointers, no other arithmetic operations can be performed on pointers. For example, you cannot add or subtract **float** or **double** values to or from pointers.

To see the effects of pointer arithmetic, execute the next short program. It prints the actual physical addresses to which an integer pointer (**ip**) and a floating-point pointer (**fp**) are pointing. Observe how each changes, relative to its referent type, each time the loop is repeated.

```
// Demonstrate the effects of pointer arithmetic.

using System;

class PtrArithDemo {
  unsafe static void Main() {
    int x;
    int i;
    double d;

    int* ip = &i;
    double* fp = &d;

    Console.WriteLine("int     double\n");

    for(x=0; x < 10; x++) {
       Console.WriteLine((uint) (ip) + " " + (uint) (fp));
       ip++;
       fp++;
    }
  }
}
```

Sample output is shown here. Your output may differ, but the intervals will be the same.

```
int     double

1243464 1243468
1243468 1243476
1243472 1243484
1243476 1243492
1243480 1243500
1243484 1243508
1243488 1243516
1243492 1243524
1243496 1243532
1243500 1243540
```

As the output shows, pointer arithmetic is performed relative to the referent type of the pointer. Since an **int** is 4 bytes and a **double** is 8 bytes, the addresses change in increments of these values.

Pointer Comparisons

Pointers can be compared using the relational operators, such as **==**, **<**, and **>**. However, for the outcome of a pointer comparison to be meaningful, usually the two pointers must have some relationship to each other. For example, if **p1** and **p2** are pointers that point to two separate and unrelated variables, then any comparison between **p1** and **p2** is generally meaningless. However, if **p1** and **p2** point to variables that are related to each other, such as elements of the same array, then **p1** and **p2** can be meaningfully compared.

Pointers and Arrays

In C#, pointers and arrays are related. For example, within a **fixed** statement, the name of an array without any index generates a pointer to the start of the array. Consider the following program:

```
/* An array name without an index yields a pointer to the
   start of the array. */

using System;

class PtrArray {
  unsafe static void Main() {
    int[] nums = new int[10];

    fixed(int* p = &nums[0], p2 = nums) {
      if(p == p2)
        Console.WriteLine("p and p2 point to same address.");
    }
  }
}
```

The output is shown here:

```
p and p2 point to same address.
```

As the output shows, the expression

```
&nums[0]
```

is the same as

```
nums
```

Since the second form is shorter, most programmers use it when a pointer to the start of an array is needed.

Indexing a Pointer

When a pointer refers to an array, the pointer can be indexed as if it were an array. This syntax provides an alternative to pointer arithmetic that can be more convenient in some situations. Here is an example:

```
// Index a pointer as if it were an array.

using System;

class PtrIndexDemo {
  unsafe static void Main() {
    int[] nums = new int[10];

    // Index a pointer.
    Console.WriteLine("Index pointer like array.");
    fixed (int* p = nums) {
      for(int i=0; i < 10; i++)
        p[i] = i; // index pointer like array

      for(int i=0; i < 10; i++)
        Console.WriteLine("p[{0}]: {1} ", i, p[i]);
    }

    // Use pointer arithmetic.
    Console.WriteLine("\nUse pointer arithmetic.");
    fixed (int* p = nums) {
      for(int i=0; i < 10; i++)
        *(p+i) = i; // use pointer arithmetic

      for(int i=0; i < 10; i++)
        Console.WriteLine("*(p+{0}): {1} ", i, *(p+i));
    }
  }
}
```

The output is shown here:

```
Index pointer like array.
p[0]: 0
p[1]: 1
p[2]: 2
p[3]: 3
```

```
p[4]: 4
p[5]: 5
p[6]: 6
p[7]: 7
p[8]: 8
p[9]: 9

Use pointer arithmetic.
*(p+0): 0
*(p+1): 1
*(p+2): 2
*(p+3): 3
*(p+4): 4
*(p+5): 5
*(p+6): 6
*(p+7): 7
*(p+8): 8
*(p+9): 9
```

As the program illustrates, a pointer expression with this general form

*(ptr + i)

can be rewritten using array-indexing syntax like this:

ptr[i]

There are two important things to understand about indexing a pointer: First, no boundary checking is applied. Thus, it is possible to access an element beyond the end of the array to which the pointer refers. Second, a pointer does not have a **Length** property. So, using the pointer, there is no way of knowing how long the array is.

Pointers and Strings

Although strings are implemented as objects in C#, it is possible to access the characters in a string through a pointer. To do so, you will assign a pointer to the start of the string to a **char*** pointer using a **fixed** statement like this:

fixed(char* *p* = *str*) { // ...

After the **fixed** statement executes, **p** will point to the start of the array of characters that make up the string. This array is *null-terminated*, which means that it ends with a zero. You can use this fact to test for the end of the array. Null-terminated character arrays are the way that strings are implemented in C/C++. Thus, obtaining a **char*** pointer to a **string** allows you to operate on strings in much the same way as does C/C++.

Here is a program that demonstrates accessing a string through a **char*** pointer:

```
// Use fixed to get a pointer to the start of a string.

using System;

class FixedString {
  unsafe static void Main() {
    string str = "this is a test";
```

```
    // Point p to start of str.
    fixed(char* p = str) {

      // Display the contents of str via p.
      for(int i=0; p[i] != 0; i++)
        Console.Write(p[i]);
    }

    Console.WriteLine();

  }
}
```

The output is shown here:

```
this is a test
```

Multiple Indirection

You can have a pointer point to another pointer that points to the target value. This situation is called *multiple indirection*, or *pointers to pointers.* Pointers to pointers can be confusing. Figure 20-1 helps clarify the concept of multiple indirection. As you can see, the value of a normal pointer is the address of the variable that contains the value desired. In the case of a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the variable that contains the value desired.

Multiple indirection can be carried on to whatever extent desired, but more than a pointer to a pointer is rarely needed. In fact, excessive indirection is difficult to follow and prone to conceptual errors.

A variable that is a pointer to a pointer must be declared as such. You do this by placing an additional asterisk after the type name. For example, the following declaration tells the compiler that **q** is a pointer to a pointer of type **int**:

```
int** q;
```

You should understand that **q** is not a pointer to an integer, but rather a pointer to an **int** pointer.

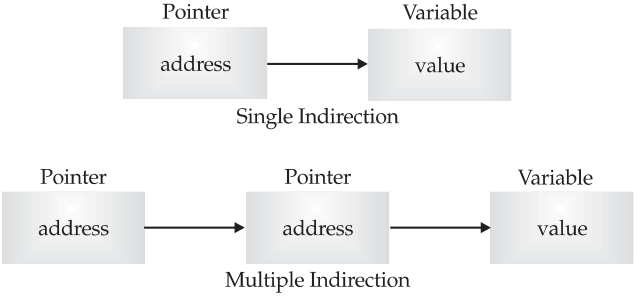


FIGURE 20-1 Single and multiple indirection

To access the target value indirectly pointed to by a pointer to a pointer, you must apply the asterisk operator twice, as in this example:

```
using System;

class MultipleIndirect {
  unsafe static void Main() {
    int x;   // holds an int value
    int* p;  // holds an int pointer
    int** q; // holds a pointer to an int pointer

    x = 10;
    p = &x; // put address of x into p
    q = &p; // put address of p into q

    Console.WriteLine(**q); // display the value of x
  }
}
```

The output is the value of **x**, which is 10. In the program, **p** is declared as a pointer to an **int** and **q** as a pointer to an **int** pointer.

One last point: Do not confuse multiple indirection with high-level data structures, such as linked lists. These are two fundamentally different concepts.

Arrays of Pointers

Pointers can be arrayed like any other data type. The declaration for an **int** pointer array of size 3 is

```
int * [] ptrs = new int * [3];
```

To assign the address of an **int** variable called **var** to the third element of the pointer array, write

```
ptrs[2] = &var;
```

To find the value of **var**, write

```
*ptrs[2]
```

sizeof

When working in an unsafe context, you might occasionally find it useful to know the size, in bytes, of one of C#'s value types. To obtain this information, use the **sizeof** operator. It has this general form:

sizeof(*type*)

Here, type is the *type* whose size is being obtained. In general, **sizeof** is intended primarily for special-case situations, especially when working with a blend of managed and unmanaged code.

stackalloc

You can allocate memory from the stack by using **stackalloc**. It can be used only when initializing local variables and has this general form:

type *p = stackalloc *type*[*size*]

Here, *p* is a pointer that receives the address of the memory that is large enough to hold *size* number of objects of *type*. Also, *type* must be a nonreference type. If there is not room on the stack to allocate the memory, a **System.StackOverflowException** is thrown. Finally, **stackalloc** can be used only in an unsafe context.

Normally, memory for objects is allocated from the *heap*, which is a region of free memory. Allocating memory from the stack is the exception. Variables allocated on the stack are not garbage-collected. Rather, they exist only while the method in which they are declared is executing. When the method is left, the memory is freed. One advantage to using **stackalloc** is that you don't need to worry about the memory being moved about by the garbage collector.

Here is an example that uses **stackalloc**:

```
// Demonstrate stackalloc.

using System;

class UseStackAlloc {
  unsafe static void Main() {
    int* ptrs = stackalloc int[3];

    ptrs[0] = 1;
    ptrs[1] = 2;
    ptrs[2] = 3;

    for(int i=0; i < 3; i++)
      Console.WriteLine(ptrs[i]);
  }
}
```

The output is shown here:

```
1
2
3
```

Creating Fixed-Size Buffers

There is a second use of the **fixed** keyword that enables you to create fixed-sized, single-dimensional arrays. In the C# documentation, these are referred to as *fixed-size buffers*. A fixed-size buffer is always a member of a **struct**. The purpose of a fixed-size buffer is to allow the creation of a **struct** in which the array elements that make up the buffer are contained within the **struct**. Normally, when you include an array member in a **struct**, only a reference to the array is actually held within the **struct**. By using a fixed-size buffer, you cause the entire array to be contained within the **struct**. This results in a structure that can be used in situations in which the size of a **struct** is important, such as in mixed-language programming, interfacing to data not created by a C# program, or whenever a nonmanaged **struct** containing an array is required. Fixed-size buffers can be used only within an unsafe context.

To create a fixed-size buffer, use this form of **fixed**:

fixed *type buf-name*[*size*];

Here, *type* is the data type of the array; *buf-name* is the name of the fixed-size buffer; and *size* is the number of elements in the buffer. Fixed-size buffers can be specified only within a **struct**.

To understand why a fixed-size buffer might be useful, consider a situation in which you want to pass bank account information to an account management program that is written in C++. Furthermore, assume that each account record uses the following organization:

Name	An 8-bit, ASCII character string, 80 bytes long
Balance	A **double**, 8 bytes long
ID	A **long**, 8 bytes long

In C++, each structure, itself, contains the **Name** array. This differs from C#, which would normally just store a reference to the array. Thus, representing this data in a C# **struct** requires the use of a fixed-size buffer, as shown here:

```
// Use a fixed-size buffer.
unsafe struct FixedBankRecord {
  public fixed byte Name[80]; // create a fixed-size buffer
  public double Balance;
  public long ID;
}
```

By using a fixed-size buffer for **Name**, each instance of **FixedBankRecord** will contain all 80 bytes of the **Name** array, which is the way that a C++ **struct** would be organized. Thus, the overall size of **FixedBankRecord** is 96, which is the sum of its members. Here is a program that demonstrates this fact:

```
// Demonstrate a fixed-size buffer.

using System;

// Create a fixed-size buffer.
unsafe struct FixedBankRecord {
  public fixed byte Name[80]; // create a fixed-size buffer
  public double Balance;
  public long ID;
}

class FixedSizeBuffer {
  // Mark Main as unsafe.
  unsafe static void Main() {
    Console.WriteLine("Size of FixedBankRecord is " +
                      sizeof(FixedBankRecord));
  }
}
```

The output is shown here:

```
Size of FixedBankRecord is 96
```

Although the size of **FixedBankRecord** is the exact sum of its members, this may not be the case for all **struct**s that have fixed-size buffers. C# is free to pad the overall length of structure so that it aligns on an even boundary (such as a word boundary) for efficiency reasons. Therefore, the overall length of a **struct** might be a few bytes greater than the sum of its fields, even when fixed-size buffers are used. In most cases, an equivalent C++ **struct** would also use the same padding. However, be aware that a difference in this regard may be possible.

One last point: In the program, notice how the fixed-size buffer for **Name** is created:

```
public fixed byte Name[80]; // create a fixed-size buffer
```

Pay special attention to how the dimension of the array is specified. The brackets containing the array size follow the array name. This is C++-style syntax, and it differs from normal C# array declarations. This statement allocates 80 bytes of storage within each **FixedBankRecord** object.

Nullable Types

Beginning with version 2.0, C# has included a feature that provides an elegant solution to what is both a common and irritating problem. The feature is the *nullable type.* The problem is how to recognize and handle fields that do not contain values (in other words, unassigned fields). To understand the problem, consider a simple customer database that keeps a record of the customer's name, address, customer ID, invoice number, and current balance. In such a situation, it is possible to create a customer entry in which one or more of those fields would be unassigned. For example, a customer may simply request a catalog. In this case, no invoice number would be needed and the field would be unused.

In the past, handling the possibility of unused fields required the use of either placeholder values or an extra field that simply indicated whether a field was in use. Of course, placeholder values could work only if there was a value that would otherwise be invalid, which won't be the case in all situations. Adding an extra field to indicate if a field is in use works in all cases, but having to manually create and manage such a field is an annoyance. The nullable type solves both problems.

Nullable Basics

A nullable type is a special version of a value type that is represented by a structure. In addition to the values defined by the underlying type, a nullable type can also store the value **null**. Thus, a nullable type has the same range and characteristics as its underlying type. It simply adds the ability to represent a value that indicates that a variable of that type is unassigned. Nullable types are objects of **System.Nullable<T>**, where **T** must be a non-nullable value type.

REMEMBER *Only value types have nullable equivalents.*

A nullable type can be specified two different ways. First, you can explicitly declare objects of type **Nullable<T>**, which is defined in the **System** namespace. For example, this creates **int** and **bool** nullable types:

```
System.Nullable<int> count;
System.Nullable<bool> done;
```

The second way to declare a nullable type is much shorter and is more commonly used. Simply follow the type name with a **?**. For example, the following shows the more common way to declare a nullable **int** and **bool** type:

```
int? count;
bool? done;
```

When using nullable types, you will often see a nullable object created like this:

```
int? count = null;
```

This explicitly initializes **count** to **null**. This satisfies the constraint that a variable must be given a value before it is used. In this case, the value simply means undefined.

You can assign a value to a nullable variable in the normal way because a conversion from the underlying type to the nullable type is predefined. For example, this assigns the value 100 to **count**.

```
count = 100;
```

There are two ways to determine if a variable of a nullable type is **null** or contains a value. First, you can test its value against **null**. For example, using **count** declared by the preceding statement, the following determines if it has a value:

```
if(count != null) // has a value
```

If **count** is not **null**, then it contains a value.

The second way to determine if a nullable type contains a value is to use the **HasValue** read-only property defined by **Nullable<T>**. It is shown here:

bool HasValue

HasValue will return true if the instance on which it is called contains a value. It will return false otherwise. Using the **HasValue** property, here is the second way to determine if the nullable object **count** has a value:

```
if(count.HasValue) // has a value
```

Assuming that a nullable object contains a value, you can obtain its value by using the **Value** read-only property defined by **Nullable<T>**, which is shown here:

T Value

It returns the value of the nullable instance on which it is called. If you try to obtain a value from a variable that is **null**, a **System.InvalidOperationException** will be thrown. It is also possible to obtain the value of a nullable instance by casting it into its underlying type.

The following program puts together the pieces and demonstrates the basic mechanism that handles a nullable type:

```
// Demonstrate a nullable type.

using System;

class NullableDemo {
  static void Main() {
```

```
    int? count = null;

    if(count.HasValue)
      Console.WriteLine("count has this value: " + count.Value);
    else
      Console.WriteLine("count has no value");

    count = 100;

    if(count.HasValue)
      Console.WriteLine("count has this value: " + count.Value);
    else
      Console.WriteLine("count has no value");
  }
}
```

The output is shown here:

```
count has no value
count has this value: 100
```

Nullable Objects in Expressions

A nullable object can be used in expressions that are valid for its underlying type. Furthermore, it is possible to mix nullable objects and non-nullable objects within the same expression. This works because of the predefined conversion that exists from the underlying type to the nullable type. When non-nullable and nullable types are mixed in an operation, the outcome is a nullable value.

The following program illustrates the use of nullable types in expressions:

```
// Use nullable objects in expressions.

using System;

class NullableDemo {
  static void Main() {
    int? count = null;
    int? result = null;

    int incr = 10; // notice that incr is a non-nullable type

    // result contains null, because count is null.
    result = count + incr;

    if(result.HasValue)
      Console.WriteLine("result has this value: " + result.Value);
    else
      Console.WriteLine("result has no value");

    // Now, count is given a value and result will contain a value.
    count = 100;
    result = count + incr;
```

```
    if(result.HasValue)
      Console.WriteLine("result has this value: " + result.Value);
    else
      Console.WriteLine("result has no value");

  }
}
```

The output is shown here:

```
result has no value
result has this value: 110
```

The ?? Operator

If you attempt to use a cast to convert a nullable object to its underlying type, a **System.InvalidOperationException** will be thrown if the nullable object contains a **null** value. This can occur, for example, when you use a cast to assign the value of a nullable object to a variable of its underlying type. You can avoid the possibility of this exception being thrown by using the **??** operator, which is called the *null coalescing operator*. It lets you specify a default value that will be used when the nullable object contains **null**. It also eliminates the need for the cast.

The **??** operator has this general form:

nullable-object ?? *default-value*

If *nullable-object* contains a value, then the value of the **??** is that value. Otherwise, the value of the **??** operation is *default-value*.

For example, in the following code **balance** is **null**. This causes **currentBalance** to be assigned the value 0.0 and no exception will be thrown.

```
double? balance = null;
double currentBalance;

currentBalance = balance ?? 0.0;
```

In the next sequence, **balance** is given the value 123.75:

```
double? balance = 123.75;
double currentBalance;

currentBalance = balance ?? 0.0;
```

Now, **currentBalance** will contain the value of **balance**, which is 123.75.

One other point: The right-hand expression of the **??** is evaluated only if the left-hand expression does not contain a value. The following program demonstrates this fact:

```
// Using ??

using System;

class NullableDemo2 {
```

```
  // Return a zero balance.
  static double GetZeroBal() {
    Console.WriteLine("In GetZeroBal().");
      return 0.0;
  }

  static void Main() {
    double? balance = 123.75;
    double currentBalance;

    // Here, GetZeroBal( ) is not called because balance
    // contains a value.
    currentBalance = balance ?? GetZeroBal();

    Console.WriteLine(currentBalance);
  }
}
```

In this program, the method **GetZeroBal( )** is not called because **balance** contains a value. As explained, when the left-hand expression of **??** contains a value, the right-hand expression is not evaluated.

Nullable Objects and the Relational and Logical Operators

Nullable objects can be used in relational expressions in just the same way as their corresponding non-nullable types. However, there is one additional rule that applies. When two nullable objects are compared using the **<**, **>**, **<=**, or **>=** operators, the result is false if either of the objects is **null**. For example, consider this sequence:

```
byte? lower = 16;
byte? upper = null;

// Here, lower is defined, but upper isn't.
if(lower < upper) // false
```

Here, the result of the test for less than is false. However, somewhat counterintuitively, so is the inverse comparison:

```
if(lower > upper) // .. also false!
```

Thus, when one (or both) of the nullable objects used in a comparison is **null**, the result of that comparison is always false. Thus, **null** does not participate in an ordering relationship.

You can test whether a nullable object contains **null**, however, by using the **==** or **!=** operators. For example, this is a valid test that will result in a true outcome:

```
if(upper == null) // ...
```

When a logical expression involves two **bool?** objects, the outcome of that expression will be one of three values: **true**, **false**, or **null** (undefined). Here are the entries that are added to the truth table for the **&** and **|** operators that apply to **bool?**.

P	Q	P | Q	P & Q
true	null	true	null
false	null	null	false
null	true	true	null
null	false	null	false
null	null	null	null

One other point: When the **!** operator is applied to a **bool?** value that is **null**, the outcome is **null**.

Partial Types

Beginning with C# 2.0, a class, structure, or interface definition can be broken into two or more pieces, with each piece residing in a separate file. This is accomplished through the use of the **partial** keyword. When your program is compiled, the pieces are united.

When used to create a partial type, the **partial** modifier has this general form:

partial *type typename* { // ...

Here, *typename* is the name of the class, structure, or interface that is being split into pieces. Each part of a partial type must be modified by **partial**.

Here is an example that divides a simple XY coordinate class into three separate files. The first file is shown here:

```
partial class XY {
  public XY(int a, int b) {
    X = a;
    Y = b;
  }
}
```

The second file is shown next:

```
partial class XY {
  public int X { get; set; }
}
```

The third file is

```
partial class XY {
  public int Y { get; set; }
}
```

The following file demonstrates the use of **XY**:

```
// Demonstrate partial class definitions.
using System;
```

```
class Test {
  static void Main() {
    XY xy = new XY(1, 2);


    Console.WriteLine(xy.X + "," + xy.Y);
  }
}
```

To use **XY**, all files must be included in the compile. For example, assuming the **XY** files are called **xy1.cs**, **xy2.cs**, and **xy3.cs**, and that the **Test** class is contained in a file called **test.cs**, then to compile **Test**, use the following command line:

```
csc test.cs xy1.cs xy2.cs xy3.cs
```

One last point: It is legal to have partial generic classes. However, the type parameters of each partial declaration must match the other parts.

Partial Methods

As the preceding section described, you can use **partial** to create a partial type. Beginning with C# 3.0, there is a second use of **partial** that lets you create a *partial method* within a partial type. A partial method has its declaration in one part and its implementation in another part. Thus, in a partial class or structure, **partial** can be used to allow the declaration of a method to be separate from its implementation.

The key aspect of a partial method is that the implementation is not required! When the partial method is not implemented by another part of the class or structure, then all calls to the partial method are silently ignored. This makes it possible for a class to specify, but not require, optional functionality. If that functionality is not implemented, then it is simply ignored.

Here is an expanded version of the preceding program that creates a partial method called **Show( )**. It is called by another method called **ShowXY( )**. (For convenience, all pieces of the partial class **XY** are shown in one file, but they could have been organized into separate files, as illustrated in the preceding section.)

```
// Demonstrate a partial method.
using System;

partial class XY {
  public XY(int a, int b) {
    X = a;
    Y = b;
  }

  // Declare a partial method.
  partial void Show();
}

partial class XY {
  public int X { get; set; }
```

```
  // Implement a partial method.
  partial void Show() {
    Console.WriteLine("{0}, {1}", X, Y);
  }
}

partial class XY {
  public int Y { get; set; }

  // Call a partial method.
  public void ShowXY() {
    Show();
  }
}

class Test {
  static void Main() {
    XY xy = new XY(1, 2);

    xy.ShowXY();
  }
}
```

Notice that **Show( )** is declared in one part of **XY** and implemented by another part. The implementation displays the values of **X** and **Y**. This means that when **Show( )** is called by **ShowXY( )**, the call has effect, and it will, indeed, display **X** and **Y**. However, if you comment-out the implementation of **Show( )**, then the call to **Show( )** within **ShowXY( )** does nothing.

Partial methods have several restrictions, including these: They must return **void**. They cannot have access modifiers. They cannot be virtual. They cannot use **out** parameters.

Friend Assemblies

It is possible to make one assembly the *friend* of another. A friend has access to the private members of the assembly of which it is a friend. This feature makes it possible to share members between selected assemblies without making those members public.

To declare a friend assembly, you must specify the friend assembly's name and its public key token in an **InternalsVisibleTo** attribute.

Miscellaneous Keywords

To conclude Part I, the few remaining keywords defined by C# that have not been described elsewhere are briefly discussed.

lock

The **lock** keyword is used when creating multithreaded programs. It is examined in detail in Chapter 23, where multithreaded programming is discussed. A brief description is given here for the sake of completeness.

In C#, a program can contain more than one *thread of execution.* When this is the case, the program is said to be *multithreaded,* and pieces of the program are executed concurrently. Thus, pieces of the program execute independently and simultaneously. This raises the prospect of a special type of problem: What if two threads try to use a resource that can be used by only one thread at a time? To solve this problem, you can create a *critical code section* that will be executed by one and only one thread at a time. This is accomplished by **lock**. Its general form is shown here:

```
lock(obj) {
  // critical section
}
```

Here, *obj* is the object on which the lock is synchronized. If one thread has already entered the critical section, then a second thread will wait until the first thread exits the critical section. When the first thread leaves the critical section, the lock is released and the second thread can be granted the lock, at which point the second thread can execute the critical section.

NOTE ***lock*** *is discussed in detail in Chapter 23.*

readonly

You can create a read-only field in a class by declaring it as **readonly**. A **readonly** field can be given a value only by using an initializer when it is declared or by assigning it a value within a constructor. Once the value has been set, it can't be changed outside the constructor. Thus, a **readonly** field is a good way to create a fixed value that has its value set by a constructor. For example, you might use a **readonly** field to represent an array dimension that is used frequently throughout a program. Both static and non-static **readonly** fields are allowed.

NOTE *Although similar,* ***readonly*** *fields are not the same as* ***const*** *fields, which are described in the following section.*

Here is an example that creates a **readonly** field:

```
// Demonstrate readonly.

using System;

class MyClass {
  public static readonly int SIZE = 10;
}

class DemoReadOnly {
  static void Main() {
    int[] source = new int[MyClass.SIZE];
    int[] target = new int[MyClass.SIZE];

    // Give source some values.
    for(int i=0; i < MyClass.SIZE; i++)
      source[i] = i;
```

```
    foreach(int i in source)
      Console.Write(i + " ");

    Console.WriteLine();

    // Reverse copy source into target.
    for(int i = MyClass.SIZE-1, j = 0; i > 0; i--, j++)
      target[j] = source[i];

    foreach(int i in target)
      Console.Write(i + " ");

    Console.WriteLine();

//  MyClass.SIZE = 100; // Error!!! can't change
  }
}
```

Here, **MyClass.SIZE** is initialized to 10. After that, it can be used, but not changed. To prove this, try removing the comment symbol from before the last line and then compiling the program. As you will see, an error will result.

const and volatile

The **const** modifier is used to declare fields or local variables that cannot be changed. These variables must be given initial values when they are declared. Thus, a **const** variable is essentially a constant. For example,

```
const int i = 10;
```

creates a **const** variable called **i** that has the value 10. Although a **const** field is similar to a **readonly** field, the two are not the same. A **const** field cannot be set within a constructor, but a **readonly** field can.

The **volatile** modifier tells the compiler that a field's value may be changed by two or more concurrently executing threads. In this situation, one thread may not know when the field has been changed by another thread. This is important because the C# compiler will automatically perform certain optimizations that work only when a field is accessed by a single thread of execution. To prevent these optimizations from being applied to a shared field, declare it **volatile**. This tells the compiler that it must obtain the value of this field each time it is accessed.

The using Statement

In addition to the **using** *directive* discussed earlier, **using** has a second form that is called the **using** *statement.* It has these general forms:

```
using (obj) {
  // use obj
}

using (type obj = initializer) {
  // use obj
}
```

Here, *obj* is an expression that must evaluate to an object that implements the **System.IDisposable** interface. It specifies a variable that will be used inside the **using** block. In the first form, the object is declared outside the **using** statement. In the second form, the object is declared within the **using** statement. When the block concludes, the **Dispose( )** method (defined by the **System.IDisposable** interface) will be called on *obj*. **Dispose( )** is called even if the **using** block ends because of an exception. Thus, a **using** statement provides a means by which objects are automatically disposed when they are no longer needed. Remember, the **using** statement applies only to objects that implement the **System.IDisposable** interface.

Here is an example of each form of the **using** statement:

```
// Demonstrate using statement.

using System;
using System.IO;

class UsingDemo {
  static void Main() {
    try {
      StreamReader sr = new StreamReader("test.txt");

      // Use object inside using statement.
      using(sr) {
        // ...
      }
    } catch(IOException exc) {
       // ...
    }

    try {
      // Create a StreamReader inside the using statement.
      using(StreamReader sr2 = new StreamReader("test.txt")) {
        // ...
      }
    } catch(IOException exc) {
      // ...
    }
  }
}
```

The class **StreamReader** implements the **IDisposable** interface (through its base class **TextReader**). Thus, it can be used in a **using** statement. When the **using** statement ends, **Dispose( )** is automatically called on the stream variable, thus closing the stream.

As the preceding example illustrates, **using** is particularly useful when working with files because the file is automatically closed at the end of the **using** block, even if the block ends because of an exception. As a result, closing a file via **using** often simplifies file-handling code.

extern

The **extern** keyword has two uses. Each is examined here.

Declaring extern Methods

The first use of **extern** has been available since the creation of C#. It indicates that a method is provided by unmanaged code that is not part of the program. In other words, that method is supplied by external code.

To declare a method as external, simply precede its declaration with the **extern** modifier. The declaration must not include any body. Thus, the general form of an **extern** declaration is as shown here:

extern *ret-type meth-name*(*arg-list*);

Notice that no braces are used.

In this use, **extern** is often used with the **DllImport** attribute, which specifies the DLL that contains the method. **DllImport** is in the **System.Runtime.InteropServices** namespace. It supports several options, but for most uses, it is sufficient to simply specify the name of the DLL that contains the **extern** method. In general, **extern** methods should be coded in C. (If you use C++, then the name of the method within the DLL might be altered with the addition of type decorations.)

To best understand how to use **extern** methods, it is helpful to work through an example. The example consists of two files. The first is the C file shown here, which defines a method called **AbsMax( )**. Call this file **ExtMeth.c**.

```
#include <stdlib.h>

int __declspec(dllexport) AbsMax(int a, int b) {
  return abs(a) < abs(b) ? abs(b) : abs(a);
}
```

The **AbsMax( )** method compares the absolute values of its two parameters and returns the maximum. Notice the use of **__declspec(dllexport)**. This is a Microsoft-specific extension to the C language that tells the compiler to export the **AbsMax( )** method within the DLL that contains it. You must use this command line to compile **ExtMeth.c.**

```
CL /LD /MD ExtMeth.c
```

This creates a DLL file called **ExtMeth.dll**.

Next is a program that uses **AbsMax( )**:

```
using System;
using System.Runtime.InteropServices;

class ExternMeth {

  // Here an extern method is declared.
  [DllImport("ExtMeth.dll")]
  public extern static int AbsMax(int a, int b);

  static void Main() {

    // Use the extern method.
```

```
    int max = AbsMax(-10, -20);
    Console.WriteLine(max);

  }
}
```

Notice the use of the **DllImport** attribute. It tells the compiler what DLL contains the **extern** method **AbsMax( )**. In this case, the file is **ExtMeth.dll**, which is the file DLL created when the C file was compiled. When the program is run, the value 20 is displayed, as expected.

Declaring an extern Assembly Alias

A second form of **extern** (which was added by C# 2.0) provides an alias for an external assembly. It is used in cases in which a program includes two separate assemblies that both contain the same name. For example, if an assembly called **test1** contains a class called **MyClass** and **test2** also contains a class called **MyClass**, then a conflict will arise if both classes need to be used within the same program.

To solve this problem, you must create an alias for each assembly. This is a two-step process. First, you must specify the aliases using the **/r** compiler option. For example:

```
/r:Asm1=test1
/r:Asm2=test2
```

Second, you must specify **extern** statements that refer to these aliases. Here is the form of **extern** that creates an assembly alias:

extern alias *assembly-name*;

Continuing the example, these lines must appear in your program:

```
extern alias Asm1;
extern alias Asm2;
```

Now, either version of **MyClass** can be accessed by qualifying it with its alias.

Here is a complete example that demonstrates an **extern** alias. It contains three files. The first is shown here. It should be put in a file called **test1.cs**.

```
using System;

namespace MyNS {
  public class MyClass {
    public MyClass() {
      Console.WriteLine("Constructing from MyClass1.dll.");
    }
  }
}
```

The second file is called **test2.cs**. It is shown here:

```
using System;

namespace MyNS {
  public class MyClass {
    public MyClass() {
      Console.WriteLine("Constructing from MyClass2.dll.");
```

```
    }
  }
}
```

Notice that both **test1.cs** and **test2.cs** define a namespace called **MyNS**, and that within that namespace, both files define a class called **MyClass**. Thus, without an **extern** alias, no program could have access to both versions of **MyClass**.

The third file, **test3.cs**, which is shown next, uses **MyClass** from both **test1.cs** and **test2.cs**. It is able to do this because of the **extern** alias statements.

```
// extern alias statements must be at the top of the file.
extern alias Asm1;
extern alias Asm2;

using System;

class Demo {
  static void Main() {
    Asm1::MyNS.MyClass t = new Asm1::MyNS.MyClass();
    Asm2::MyNS.MyClass t2 = new Asm2::MyNS.MyClass();
  }
}
```

Start by compiling **test1.cs** and **test2.cs** into DLLs. This can be easily done from the command line by using these commands:

```
csc /t:library test1.cs
csc /t:library test2.cs
```

Next, compile **test3.cs** by using this command line:

```
csc /r:Asm1=test1.dll /r:Asm2=test2.dll test3.cs
```

Notice the use of the **/r** option, which tells the compiler to reference the metadata found in the associated file. In this case, the alias **Asm1** is linked with **test1.dll** and the alias **Asm2** is linked with **test2.dll**.

Within the program, the aliases are specified by these two **extern** statements at the top of the file:

```
extern alias Asm1;
extern alias Asm2;
```

Within **Main( )**, the aliases are used to disambiguate the references to **MyClass**. Notice how the alias is used to refer to **MyClass**:

```
Asm1::MyNS.MyClass
```

The alias is specified first, followed by the namespace resolution operator, followed by the name of the namespace that contains the ambiguous class, followed by the dot operator and the class name. This same general form works with other **extern** aliases.

The output from the program is shown here:

```
Constructing from MyClass1.dll.
Constructing from MyClass2.dll.
```

This page intentionally left blank

PART

Exploring the C# Library

CHAPTER 21
Exploring the System Namespace

CHAPTER 22
Strings and Formatting

CHAPTER 23
Multithreaded Programming

CHAPTER 24
Collections, Enumerators, and Iterators

CHAPTER 25
Networking Through the Internet Using System.Net

CHAPTER 26
Use System.Windows.Forms to Create Form-Based Windows Applications

Part II explores the C# library. As explained in Part I, the class library used by C# is the .NET Framework class library. As a result, the material in this section applies not only to C#, but to the .NET Framework as a whole.

The .NET Framework class library is organized into namespaces. To use a portion of the library, you will normally import its namespace by including a **using** directive. Of course, you can also fully qualify the name of the item with its namespace name, but most often, it is simply easier to import the entire namespace.

The .NET library is very large, and it is beyond the scope of this book to examine each part of it. (A complete description would easily fill a very large book!) Instead, Part II examines the core elements of the library, many of which are contained in the **System** namespace. Also discussed are the collection classes, multithreading, and networking.

NOTE *The I/O classes are discussed in Chapter 14.*

This page intentionally left blank

CHAPTER 21

Exploring the System Namespace

This chapter explores the **System** namespace. **System** is a top-level namespace of the .NET Framework class library. It directly contains those classes, structures, interfaces, delegates, and enumerations that are most commonly used by a C# program or that are deemed otherwise integral to the .NET Framework. Thus, **System** defines the core of the library.

System also contains many nested namespaces that support specific subsystems, such as **System.Net**. Several of these subsystems are described later in this book. This chapter is concerned only with the members of **System**, itself.

The Members of System

In addition to a large number of exception classes, **System** contains the following classes:

ActivationContext	Activator	AppDomain
AppDomainManager	AppDomainSetup	ApplicationId
ApplicationIdentity	Array	AssemblyLoadEventArgs
Attribute	AttributeUsageAttribute	BitConverter
Buffer	CharEnumerator	CLSCompliantAttribute
Console	ConsoleCancelEventArgs	ContextBoundObject
ContextStaticAttribute	Convert	DBNull
Delegate	Enum	Environment
EventArgs	Exception	FileStyleUriParser
FlagsAttribute	FtpStyleUriParser	GC
GenericUriParser	GopherStyleUriParser	HttpStyleUriParser
LdapStyleUriParser	LoaderOptimizationAttribute	LocalDataStoreSlot
MarshalByRefObject	Math	MTAThreadAttribute
MulticastDelegate	NetPipeStyleUriParser	NetTcpStyleUriParser

NewsStyleUriParser	NonSerializedAttribute	Nullable
Object	ObsoleteAttribute	OperatingSystem
ParamArrayAttribute	Random	ResolveEventArgs
SerializableAttribute	STAThreadAttribute	String
StringComparer	ThreadStaticAttribute	TimeZone
TimeZoneInfo	TimeZoneInfo.AdjustmentRule	Type
UnhandledExceptionEventArgs	Uri	UriBuilder
UriParser	UriTemplate	UriTemplateEquivalenceComparer
UriTemplateTable	UriTypeConverter	ValueType
Version	WeakReference	

System defines the following structures:

ArgIterator	ArraySegment<T>	Boolean
Byte	Char	ConsoleKeyInfo
DateTime	DateTimeOffset	Decimal
Double	Guid	Int16
Int32	Int64	IntPtr
ModuleHandle	Nullable<T>	RuntimeArgumentHandle
RuntimeFieldHandle	RuntimeMethodHandle	RuntimeTypeHandle
Sbyte	Single	TimeSpan
TimeZoneInfo.TransitionTime	TypedReference	UInt16
UInt32	UInt64	UIntPtr
Void		

System defines the following interfaces:

_AppDomain	IAppDomainSetup	IAsyncResult
ICloneable	IComparable	IComparable<T>
IConvertible	ICustomFormatter	IDisposable
IEquatable<T>	IFormatProvider	IFormattable
IServiceProvider		

System defines the following delegates:

Action	Action<T>	Action<T1, T2>
Action<T1, T2, T3>	Action<T1, T2, T3, T4>	AppDomainInitializer
AssemblyLoadEventHandler	AsyncCallback	Comparison<T>
ConsoleCancelEventHandler	Converter<T, V>	CrossAppDomainDelegate
EventHandler	EventHandler<T>	Func<TResult>
Func<T, TResult>	Func<T1, T2, TResult>	Func<T1, T2, T3, TResult>
Func<T1, T2, T3. T4, TResult>	Predicate<T>	ResolveEventHandler
UnhandledExceptionEventHandler		

System defines these enumerations:

ActivationContext.contextForm	AppDomainManagerInitializationOptions	AttributeTargets
Base64FormattingOptions	ConsoleColor	ConsoleKey
ConsoleModifiers	ConsoleSpecialKey	DateTimeKind
DayOfWeek	Environment.SpecialFolder	EnvironmentVariableTarget
GCCollectionMode	GenericUriParserOptions	LoaderOptimization
MidpointRounding	PlatformID	StringComparison
StringSplitOptions	TypeCode	UriComponents
UriFormat	UriHostNameType	UriIdnScope
UriKind	UriPartial	

As the preceding tables show, **System** is quite large. It is not possible to examine all of its constituents in detail in a single chapter. Furthermore, several of **System**'s members, such as **Nullable<T>**, **Type**, **Exception**, and **Attribute**, are discussed in Part I or elsewhere in Part II. Finally, because **System**.**String**, which defines the C# **string** type, is such a large and important topic, it is covered in Chapter 22 along with formatting. For these reasons, this chapter explores only those members that are most commonly used by C# programmers and that are not fully covered elsewhere.

The Math Class

Math defines several standard mathematical operations, such as square root, sine, cosine, and logarithms. The **Math** class is **static**, which means all of the methods defined by **Math** are **static** and no object of type **Math** can be constructed. It also means **Math** is implicitly sealed and cannot be inherited. The methods defined by **Math** are shown in Table 21-1. All angles are in radians.

Math also defines these two fields:

```
public const double E
public const double PI
```

E is the value of the natural logarithm base, commonly referred to as *e*. **PI** is the value of pi.

Method	Meaning
public static double Abs(double *v*)	Returns the absolute value of *v*.
public static float Abs(float *v*)	Returns the absolute value of *v*.
public static decimal Abs(decimal *v*)	Returns the absolute value of *v*.
public static int Abs(int *v*)	Returns the absolute value of *v*.
public static short Abs(short *v*)	Returns the absolute value of *v*.
public static long Abs(long *v*)	Returns the absolute value of *v*.
public static sbyte Abs(sbyte *v*)	Returns the absolute value of *v*.
public static double Acos(double *v*)	Returns the arc cosine of *v*. The value of *v* must be between –1 and 1.
public static double Asin(double *v*)	Returns the arc sine of *v*. The value of *v* must be between –1 and 1.
public static double Atan(double *v*)	Returns the arc tangent of *v*.
public static double Atan2(double *y*, double *x*)	Returns the arc tangent of *y*/*x*.
public static long BigMul(int *x*, int *y*)	Returns the result of *x* * *y* as a **long** value, thus avoiding overflow.
public static double Ceiling(double *v*)	Returns the smallest integer (represented as a floating-point value) not less than *v*. For example, given 1.02, **Ceiling( )** returns 2.0. Given –1.02, **Ceiling( )** returns –1.
public static double Ceiling(decimal *v*)	Returns the smallest integer (represented as a decimal value) not less than *v*. For example, given 1.02, **Ceiling( )** returns 2.0. Given –1.02, **Ceiling( )** returns –1.
public static double Cos(double *v*)	Returns the cosine of *v*.
public static double Cosh(double *v*)	Returns the hyperbolic cosine of *v*.
public static int DivRem(int *x*, int *y*, out int *rem*)	Return the result of *x* / *y*. The remainder is returned in *rem*.
public static long DivRem(long *x*, long *y*, out long *rem*)	Return the result of *x* / *y*. The remainder is returned in *rem*.
public static double Exp(double *v*)	Returns the natural logarithm base *e* raised to the *v* power.
public static decimal Floor(decimal *v*)	Returns the largest integer (represented as a decimal value) not greater than *v*. For example, given 1.02, **Floor( )** returns 1.0. Given –1.02, **Floor( )** returns –2.

TABLE 21-1 Methods Defined by **Math**

Method	Meaning
public static double Floor(double *v*)	Returns the largest integer (represented as a floating-point value) not greater than *v*. For example, given 1.02, **Floor( )** returns 1.0. Given –1.02, **Floor( )** returns –2.
public static double IEEERemainder(double *dividend*, double *divisor*)	Returns the remainder of *dividend* / *divisor*.
public static double Log(double *v*)	Returns the natural logarithm for *v*.
public static double Log(double *v*, double *base*)	Returns the logarithm for *v* using base *base*.
public static double Log10(double *v*)	Returns the base 10 logarithm for *v*.
public static double Max(double *v1*, double *v2*)	Returns the greater of *v1* and *v2*.
public static float Max(float *v1*, float *v2*)	Returns the greater of *v1* and *v2*.
public static decimal Max(decimal *v1*, decimal *v2*)	Returns the greater of *v1* and *v2*.
public static int Max(int *v1*, int *v2*)	Returns the greater of *v1* and *v2*.
public static short Max(short *v1*, short *v2*)	Returns the greater of *v1* and *v2*.
public static long Max(long *v1*, long *v2*)	Returns the greater of *v1* and *v2*.
public static uint Max(uint *v1*, uint *v2*)	Returns the greater of *v1* and *v2*.
public static ushort Max(ushort *v1*, ushort *v2*)	Returns the greater of *v1* and *v2*.
public static ulong Max(ulong *v1*, ulong *v2*)	Returns the greater of *v1* and *v2*.
public static byte Max(byte *v1*, byte *v2*)	Returns the greater of *v1* and *v2*.
public static sbyte Max(sbyte *v1*, sbyte *v2*)	Returns the greater of *v1* and *v2*.
public static double Min(double *v1*, double *v2*)	Returns the lesser of *v1* and *v2*.
public static float Min(float *v1*, float *v2*)	Returns the lesser of *v1* and *v2*.
public static decimal Min(decimal *v1*, decimal *v2*)	Returns the lesser of *v1* and *v2*.
public static int Min(int *v1*, int *v2*)	Returns the lesser of *v1* and *v2*.
public static short Min(short *v1*, short *v2*)	Returns the lesser of *v1* and *v2*.
public static long Min(long *v1*, long *v2*)	Returns the lesser of *v1* and *v2*.
public static uint Min(uint *v1*, uint *v2*)	Returns the lesser of *v1* and *v2*.
public static ushort Min(ushort *v1*, ushort *v2*)	Returns the lesser of *v1* and *v2*.

Table 21-1 Methods Defined by **Math** *(continued)*

Method	Meaning
public static ulong Min(ulong *v1*, ulong *v2*)	Returns the lesser of *v1* and *v2*.
public static byte Min(byte *v1*, byte *v2*)	Returns the lesser of *v1* and *v2*.
public static sbyte Min(sbyte *v1*, sbyte *v2*)	Returns the lesser of *v1* and *v2*.
public static double Pow(double *base*, double *exp*)	Returns *base* raised to the *exp* power($base^{exp}$).
public static double Round(double *v*)	Returns *v* rounded to the nearest whole number.
public static decimal Round(decimal *v*)	Returns *v* rounded to the nearest whole number.
public static double Round(double *v*, int *frac*)	Returns *v* rounded to the number of fractional digits specified by *frac*.
public static decimal Round(decimal *v*, int *frac*)	Returns *v* rounded to the number of fractional digits specified by *frac*.
public static double Round(double *v*, MidpointRounding *how*)	Returns *v* rounded to the nearest whole number using the rounding mode specified by *how*.
public static decimal Round(decimal *v*, MidpointRounding *how*)	Returns *v* rounded to the nearest whole number using the rounding mode specified by *how*.
public static double Round(double *v*, int *frac*, MidpointRounding *how*)	Returns *v* rounded to the number of fractional digits specified by *frac*. It uses the rounding mode specified by *how*.
public static decimal Round(decimal *v*, int *frac*, MidpointRounding *how*)	Returns *v* rounded to the number of fractional digits specified by *frac*. It uses the rounding mode specified by *how*.
public static int Sign(double *v*)	Returns –1 if *v* is less than zero, 0 if *v* is zero, and 1 if *v* is greater than zero.
public static int Sign(float *v*)	Returns –1 if *v* is less than zero, 0 if *v* is zero, and 1 if *v* is greater than zero.
public static int Sign(decimal *v*)	Returns –1 if *v* is less than zero, 0 if *v* is zero, and 1 if *v* is greater than zero.
public static int Sign(int *v*)	Returns –1 if *v* is less than zero, 0 if *v* is zero, and 1 if *v* is greater than zero.
public static int Sign(short *v*)	Returns –1 if *v* is less than zero, 0 if *v* is zero, and 1 if *v* is greater than zero.
public static int Sign(long *v*)	Returns –1 if *v* is less than zero, 0 if *v* is zero, and 1 if *v* is greater than zero.
public static int Sign(sbyte *v*)	Returns –1 if *v* is less than zero, 0 if *v* is zero, and 1 if *v* is greater than zero.
public static double Sin(double *v*)	Returns the sine of *v*.
public static double Sinh(double *v*)	Returns the hyperbolic sine of *v*.

Table 21-1 Methods Defined by **Math** *(continued)*

Method	Meaning
public static double Sqrt(double *v*)	Returns the square root of *v*.
public static double Tan(double *v*)	Returns the tangent of *v*.
public static double Tanh(double *v*)	Returns the hyperbolic tangent of *v*.
public static double Truncate(double *v*)	Returns the whole number portion of *v*.
public static decimal Truncate(decimal *v*)	Returns the whole number portion of *v*.

TABLE 21-1 Methods Defined by **Math** *(continued)*

Here is an example that uses **Sqrt( )** to help implement the Pythagorean theorem. It computes the length of the hypotenuse given the lengths of the two opposing sides of a right triangle.

```
// Implement the Pythagorean Theorem.

using System;

class Pythagorean {
  static void Main() {
    double s1;
    double s2;
    double hypot;
    string str;

    Console.WriteLine("Enter length of first side: ");
    str = Console.ReadLine();
    s1 = Double.Parse(str);

    Console.WriteLine("Enter length of second side: ");
    str = Console.ReadLine();
    s2 = Double.Parse(str);

    hypot = Math.Sqrt(s1*s1 + s2*s2);

    Console.WriteLine("Hypotenuse is " + hypot);
  }
}
```

Here is a sample run:

```
Enter length of first side: 3
Enter length of second side: 4
Hypotenuse is 5
```

Next is an example that uses the **Pow( )** method to compute the initial investment required to achieve a desired future value given the annual rate of return and the number of years. The formula to compute the initial investment is shown here:

$$InitialInvestment = FutureValue / (1 + InterestRate)^{Years}$$

Because **Pow( )** requires **double** arguments, the interest rate and the number of years are held in **double** values. The future value and initial investment use the **decimal** type.

```
/* Compute the initial investment needed to attain
   a known future value given annual rate of return
   and the time period in years. */

using System;

class InitialInvestment {
  static void Main() {
    decimal initInvest; // initial investment
    decimal futVal;     // future value

    double numYears;    // number of years
    double intRate;     // annual rate of return as a decimal

    string str;

    Console.Write("Enter future value: ");
    str = Console.ReadLine();
    try {
      futVal = Decimal.Parse(str);
    } catch(FormatException exc) {
      Console.WriteLine(exc.Message);
      return;
    }

    Console.Write("Enter interest rate (such as 0.085): ");
    str = Console.ReadLine();
    try {
      intRate = Double.Parse(str);
    } catch(FormatException exc) {
      Console.WriteLine(exc.Message);
      return;
    }

    Console.Write("Enter number of years: ");
    str = Console.ReadLine();
    try {
      numYears = Double.Parse(str);
    } catch(FormatException exc) {
      Console.WriteLine(exc.Message);
      return;
    }

    initInvest = futVal / (decimal) Math.Pow(intRate+1.0, numYears);

    Console.WriteLine("Initial investment required: {0:C}",
                      initInvest);
  }
}
```

Here is a sample run:

```
Enter future value: 10000
Enter interest rate (such as 0.085): 0.07
Enter number of years: 10
Initial investment required: $5,083.49
```

The .NET Structures Corresponding to the Built-in Value Types

The structures that correspond to C#'s built-in value types were introduced in Chapter 14 when they were used to convert strings holding human-readable numeric values into their equivalent binary values. Here these structures are examined in detail.

The .NET structure names and their C# keyword equivalents are shown in the following table:

.NET Structure Name	C# Name
System.Boolean	bool
System.Char	char
System.Decimal	decimal
System.Double	double
System.Single	float
System.Int16	short
System.Int32	int
System.Int64	long
System.UInt16	ushort
System.UInt32	uint
System.UInt64	ulong
System.Byte	byte
System.SByte	sbyte

By using the members defined by these structures, you can perform operations relating to the value types. The following sections examine each of these structures.

NOTE *Some methods defined by the structures that correspond to the built-in value types take a parameter of type* ***IFormatProvider*** *or* ***NumberStyles****.* ***IFormatProvider*** *is briefly described later in this chapter.* ***NumberStyles*** *is an enumeration found in the* ***System.Globalization*** *namespace. The topic of formatting is discussed in Chapter 22.*

The Integer Structures

The integer structures are

Byte	SByte	Int16	UInt16
Int32	UInt32	Int64	UInt64

Each of these structures contains the same methods. They are shown in Table 21-2. The only difference from structure to structure is the return type of **Parse( )**. For each structure, **Parse( )** returns a value of the type represented by the structure. For example, for **Int32**, **Parse( )** returns an **int** value. For **UInt16**, **Parse( )** returns a **ushort** value. For an example that demonstrates **Parse( )**, see Chapter 14.

In addition to the methods shown in Table 21-2, the integer structures also define the following **const** fields:

MaxValue
MinValue

For each structure, these fields contain the largest and smallest value that type of integer can hold.

All of the integer structures implement the following interfaces: **IComparable**, **IComparable<T>**, **IConvertible**, **IFormattable**, and **IEquatable<T>**, where **T** is replaced by the corresponding data type. For example, **T** will be replaced with **int** for **Int32**.

Method	Meaning
public int CompareTo(object *v*)	Compares the numerical value of the invoking object with that of *v*. Returns zero if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
public int CompareTo(*type v*)	Compares the numerical value of the invoking object with that of *v*. Returns zero if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value. In this version of **CompareTo( )**, *type* explicitly specifies the data type, such as in **System.Int32.CompareTo(int *v*)**.
public override bool Equals(object *v*)	Returns true if the value of the invoking object equals the value of *v*.
public bool Equals(*type v*)	Returns true if the value of the invoking object equals the value of *v*. In this version of **Equals( )**, *type* explicitly specifies the data type, such as in **System.Int32.Equals(int *v*)**.
public override int GetHashCode( )	Returns the hash code for the invoking object.
public TypeCode GetTypeCode( )	Returns the **TypeCode** enumeration value for the equivalent value type. For example, for **Int32**, the type code is **TypeCode.Int32**.
public static *retType* Parse(string *str*)	Returns the binary equivalent of the numeric string in *str*. If the string does not represent a numeric value as defined by the structure type, an exception is thrown. *retType* is a placeholder for the actual type of data returned based on which numeric structure is used. For example, for **Int32**, *retType* will be **int**.

Table 21-2 Methods Supported by the Integer Structures

Method	Meaning
public static *retType* Parse(string *str*, IFormatProvider *fmtpvdr*)	Returns the binary equivalent of the numeric string in *str* using the culture-specific information provided by *fmtpvdr.* If the string does not represent a numeric value as defined by the structure type, an exception is thrown. *retType* is a placeholder for the actual type of data returned based on which numeric structure is used. For example, for **Int32**, *retType* will be **int**.
public static *retType* Parse(string *str*, NumberStyles *styles*)	Returns the binary equivalent of the numeric string in *str* using the style information provided by *styles.* If the string does not represent a numeric value as defined by the structure type, an exception is thrown. *retType* is a placeholder for the actual type of data returned based on which numeric structure is used. For example, for **Int32**, *retType* will be **int**.
public static *retType* Parse(string *str*, NumberStyles *styles*, IFormatProvider *fmtpvdr*)	Returns the binary equivalent of the numeric string in *str* using the style information provided by *styles* and the culture-specific format information provided by *fmtpvdr.* If the string does not represent a numeric value as defined by structure type, an exception is thrown. *retType* is a placeholder for the actual type of data returned based on which numeric structure is used. For example, for **Int32**, *retType* will be **int**.
public override string ToString( )	Returns the string representation of the value of the invoking object.
public string ToString(string *format*)	Returns the string representation of the value of the invoking object as specified by the format string passed in *format.*
public string ToString(IFormatProvider *fmtpvdr*)	Returns the string representation of the value of the invoking object using the culture-specific information specified in *fmtpvdr.*
public string ToString(string *format*, IFormatProvider *fmtpvdr*)	Returns the string representation of the value of the invoking object using the culture-specific information specified in *fmtpvdr* and the format specified by *format.*
public static bool TryParse(string *str*, out *type val*)	Attempts to convert the numeric string in *str* into a binary value. If successful, the value is stored in *val* and true is returned. If no conversion takes place, false is returned. This differs from **Parse( )**, which throws an exception on failure. In **TryParse( )**, *type* explicitly specifies the data type, such as in **System.Int32.TryParse(int *v*)**.

Table 21-2 Methods Supported by the Integer Structures *(continued)*

Method	Meaning
public static bool TryParse(string *str*, NumberStyles *styles*, IFormatProvider *fmtpvdr*, out *type val*)	Attempts to convert the numeric string in *str* into a binary value using the style information provided by *styles* and the culture-specific format information provided by *fmtpvdr.* If successful, the value is stored in *val* and true is returned. If no conversion takes place, false is returned. This differs from **Parse( )**, which throws an exception on failure. In **TryParse( )**, *type* explicitly specifies the data type, such as in **System.Int32.TryParse(int *v*)**.

TABLE 21-2 Methods Supported by the Integer Structures *(continued)*

The Floating-Point Structures

There are two floating-point structures: **Double** and **Single**. **Single** represents **float**. Its methods are shown in Table 21-3, and its fields are shown in Table 21-4. **Double** represents **double**. Its methods are shown in Table 21-5, and its fields are shown in Table 21-6. As is the case with the integer structures, you can specify culture-specific information and format information in a call to **Parse( )** or **ToString( )**.

The floating-point structures implement the following interfaces: **IComparable**, **IComparable<T>**, **IConvertible**, **IFormattable**, and **IEquatable<T>**, where **T** is replaced by either **double** for **Double** or **float** for **Single**.

Method	Meaning
public int CompareTo(object *v*)	Compares the numerical value of the invoking object with that of *v.* Returns zero if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
public int CompareTo(float *v*)	Compares the numerical value of the invoking object with that of *v.* Returns zero if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
public override bool Equals(object *v*)	Returns true if the value of the invoking object equals the value of *v.*
public bool Equals(float *v*)	Returns true if the value of the invoking object equals the value of *v.*
public override int GetHashCode( )	Returns the hash code for the invoking object.
public TypeCode GetTypeCode( )	Returns the **TypeCode** enumeration value for **Single**, which is **TypeCode.Single**.
public static bool IsInfinity(float *v*)	Returns true if *v* represents infinity (either positive or negative). Otherwise, returns false.
public static bool IsNaN(float *v*)	Returns true if *v* is not a number. Otherwise, returns false.

TABLE 21-3 Methods Supported by **Single**

Method	Meaning
public static bool IsPositiveInfinity(float *v*)	Returns true if *v* represents positive infinity. Otherwise, returns false.
public static bool IsNegativeInfinity(float *v*)	Returns true if *v* represents negative infinity. Otherwise, returns false.
public static float Parse(string *str*)	Returns the binary equivalent of the numeric string in *str*. If the string does not represent a **float** value, an exception is thrown.
public static float Parse(string *str*, IFormatProvider *fmtpvdr*)	Returns the binary equivalent of the numeric string in *str* using the culture-specific information provided by *fmtpvdr*. If the string does not represent a **float** value, an exception is thrown.
public static float Parse(string *str*, NumberStyles *styles*)	Returns the binary equivalent of the numeric string in *str* using the style information provided by *styles*. If the string does not represent a **float** value, an exception is thrown.
public static float Parse(string *str*, NumberStyles *styles*, IFormatProvider *fmtpvdr*)	Returns the binary equivalent of the numeric string in *str* using the style information provided by *styles* and the culture-specific format information provided by *fmtpvdr*. If the string does not represent a **float** value, an exception is thrown.
public override string ToString( )	Returns the string representation of the value of the invoking object in the default format.
public string ToString(string *format*)	Returns the string representation of the value of the invoking object as specified by the format string passed in *format*.
public string ToString(IFormatProvider *fmtpvdr*)	Returns the string representation of the value of the invoking object using the culture-specific information specified in *fmtpvdr*.
public string ToString(string *format*, IFormatProvider *fmtpvdr*)	Returns the string representation of the value of the invoking object using the culture-specific information specified in *fmtpvdr* and the format specified by *format*.
public static bool TryParse(string *str*, out float *val*)	Attempts to convert the numeric string in *str* into a **float** value. If successful, the value is stored in *val* and true is returned. If no conversion takes place, false is returned. This differs from **Parse( )**, which throws an exception on failure.
public static bool TryParse(string *str*, NumberStyles *styles*, IFormatProvider *fmtpvdr*, out float *val*)	Attempts to convert the numeric string in *str* into a **float** value using the style information provided by *styles* and the culture-specific format information provided by *fmtpvdr*. If successful, the value is stored in *val* and **true** is returned. If no conversion takes place, **false** is returned. This differs from **Parse( )**, which throws an exception on failure.

TABLE 21-3 Methods Supported by **Single** *(continued)*

Field	Meaning
public const float Epsilon	The smallest non-zero positive value.
public const float MaxValue	The largest value that a **float** can hold.
public const float MinValue	The smallest value that a **float** can hold.
public const float NaN	A value that is not a number.
public const float NegativeInfinity	A value representing negative infinity.
public const float PositiveInfinity	A value representing positive infinity.

Table 21-4 Fields Supported by **Single**

Method	Meaning
public int CompareTo(object *v*)	Compares the numerical value of the invoking object with that of *v*. Returns zero if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
public int CompareTo(double *v*)	Compares the numerical value of the invoking object with that of *v*. Returns zero if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
public override bool Equals(object *v*)	Returns true if the value of the invoking object equals the value of *v*.
public bool Equals(double *v*)	Returns true if the value of the invoking object equals the value of *v*.
public override int GetHashCode( )	Returns the hash code for the invoking object.
public TypeCode GetTypeCode( )	Returns the **TypeCode** enumeration value for **Double**, which is **TypeCode.Double**.
public static bool IsInfinity(double *v*)	Returns true if *v* represents infinity (either positive or negative). Otherwise, returns false.
public static bool IsNaN(double *v*)	Returns true if *v* is not a number. Otherwise, returns false.
public static bool IsPositiveInfinity(double *v*)	Returns true if *v* represents positive infinity. Otherwise, returns false.
public static bool IsNegativeInfinity(double *v*)	Returns true if *v* represents negative infinity. Otherwise, returns false.
public static double Parse(string *str*)	Returns the binary equivalent of the numeric string in *str*. If the string does not represent a **double** value, an exception is thrown.

Table 21-5 Methods Supported by **Double**

Method	Meaning
public static double Parse(string *str*, IFormatProvider *fmtpvdr*)	Returns the binary equivalent of the numeric string in *str* using the culture-specific information provided by *fmtpvdr.* If the string does not represent a **double** value, an exception is thrown.
public static double Parse(string *str*, NumberStyles *styles*)	Returns the binary equivalent of the numeric string in *str* using the style information provided by *styles.* If the string does not represent a **double** value, an exception is thrown.
public static double Parse(string *str*, NumberStyles *styles*, IFormatProvider *fmtpvdr*)	Returns the binary equivalent of the numeric string in *str* using the style information provided by *styles* and the culture-specific format information provided by *fmtpvdr.* If the string does not represent a **double** value, an exception is thrown.
public override string ToString( )	Returns the string representation of the value of the invoking object in the default format.
public string ToString(string *format*)	Returns the string representation of the value of the invoking object as specified by the format string passed in *format.*
public string ToString(IFormatProvider *fmtpvdr*)	Returns the string representation of the value of the invoking object using the culture-specific information specified in *fmtpvdr.*
public string ToString(string *format*, IFormatProvider *fmtpvdr*)	Returns the string representation of the value of the invoking object using the culture-specific information specified in *fmtpvdr* and the format specified by *format.*
public static bool TryParse(string *str*, out double *val*)	Attempts to convert the numeric string in *str* into a **double** value. If successful, the value is stored in *val* and true is returned. If no conversion takes place, false is returned. This differs from **Parse( )**, which throws an exception on failure.
public static bool TryParse(string *str*, NumberStyles *styles,* IFormatProvider *fmtpvdr,* out double *val*)	Attempts to convert the numeric string in *str* into a **double** value using the style information provided by *styles* and the culture-specific format information provided by *fmtpvdr.* If successful, the value is stored in *val* and true is returned. If no conversion takes place, false is returned. This differs from **Parse( )**, which throws an exception on failure.

Table 21-5 Methods Supported by **Double** *(continued)*

Field	Meaning
public const double Epsilon	The smallest non-zero positive value.
public const double MaxValue	The largest value that a **double** can hold.
public const double MinValue	The smallest value that a **double** can hold.
public const double NaN	A value that is not a number.
public const double NegativeInfinity	A value representing negative infinity.
public const double PositiveInfinity	A value representing positive infinity.

TABLE 21-6 Fields Supported by **Double**

Decimal

The **Decimal** structure is a bit more complicated than its integer and floating-point relatives. It contains many constructors, fields, methods, and operators that help integrate **decimal** with the other numeric types supported by C#. For example, several of the methods provide conversions between **decimal** and the other numeric types.

Decimal offers eight public constructors. The following six are the most commonly used:

public Decimal(int *v*)
public Decimal(uint *v*)
public Decimal(long *v*)
public Decimal(ulong *v*)
public Decimal(float *v*)
public Decimal(double *v*)

Each constructs a **Decimal** from the specified value.

You can also construct a **Decimal** by specifying its constituent parts using this constructor:

public Decimal(int *low*, int *middle*, int *high*, bool *signFlag*, byte *scaleFactor*)

A decimal value consists of three parts. The first is a 96-bit integer, the second is a sign flag, and the third is a scaling factor. The 96-bit integer is passed in 32-bit chunks through *low, middle,* and *high.* The sign is passed through *signFlag,* which is **false** for a positive number and **true** for a negative number. The scaling factor is passed in *scaleFactor,* which must be a value between 0 and 28. This factor specifies the power of 10 (that is, $10^{\text{scaleFactor}}$) by which the number is divided, thus yielding its fractional component.

Instead of passing each component separately, you can specify the constituents of a **Decimal** in an array of integers, using this constructor:

public Decimal(int[ ] *parts*)

The first three **int**s in *parts* contain the 96-bit integer value. In *parts*[3], bit 31 specifies the sign flag (0 for positive, 1 for negative), and bits 16 through 23 contain the scale factor.

Decimal implements the following interfaces: **IComparable**, **IComparable<decimal>**, **IConvertible**, **IFormattable**, and **IEquatable<decimal>**.

Here is an example that constructs a **decimal** value by hand:

```
// Manually create a decimal number.
```

```
using System;

class CreateDec {
  static void Main() {
    decimal d = new decimal(12345, 0, 0, false, 2);

    Console.WriteLine(d);
  }
}
```

The output is shown here:

```
123.45
```

In this example, the value of the 96-bit integer is 12345. Its sign is positive, and it has two decimal fractions.

The methods defined by **Decimal** are shown in Table 21-7. The fields defined by **Decimal** are shown in Table 21-8. **Decimal** also defines a large number of operators and conversions that allow **decimal** values to be used in expressions with other numeric types. The rules governing the use of **decimal** in expressions and assignments are described in Chapter 3.

Method	Meaning
public static decimal Add(decimal *v1*, decimal *v2*)	Returns *v1* + *v2*.
public static decimal Ceiling(decimal *v*)	Returns the smallest integer (represented as a **decimal** value) not less than *v*. For example, given 1.02, **Ceiling( )** returns 2.0. Given –1.02, **Ceiling( )** returns –1.
public static int Compare(decimal *v1*, decimal *v2*)	Compares the numerical value of *v1* with that of *v2*. Returns zero if the values are equal. Returns a negative value if *v1* is less than *v2*. Returns a positive value if *v1* is greater than *v2*.
public int CompareTo(object *v*)	Compares the numerical value of the invoking object with that of *v*. Returns zero if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
public int CompareTo(decimal *v*)	Compares the numerical value of the invoking object with that of *v*. Returns zero if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
public static decimal Divide(decimal *v1*, decimal *v2*)	Returns *v1* / *v2*.
public bool Equals(decimal *v*)	Returns true if the value of the invoking object equals the value of *v*.

Table 21-7 Methods Defined by **Decimal**

Method	Meaning
public override bool Equals(object *v*)	Returns true if the value of the invoking object equals the value of *v*.
public static bool Equals(decimal *v1*, decimal *v2*)	Returns true if *v1* equals *v2*.
public static decimal Floor(decimal *v*)	Returns the largest integer (represented as a **decimal** value) not greater than *v*. For example, given 1.02, **Floor( )** returns 1.0. Given –1.02, **Floor( )** returns –2.
public static decimal FromOACurrency(long *v*)	Converts the OLE Automation currency value in *v* into its **decimal** equivalent and returns the result.
public static int[ ] GetBits(decimal *v*)	Returns the binary representation of *v* as an array of **int**. The organization of this array is as described in the text.
public override int GetHashCode( )	Returns the hash code for the invoking object.
public TypeCode GetTypeCode( )	Returns the **TypeCode** enumeration value for **Decimal**, which is **TypeCode.Decimal**.
public static decimal Multiply(decimal *v1*, decimal *v2*)	Returns *v1* * *v2*.
public static decimal Negate(decimal *v*)	Returns –*v*.
public static decimal Parse(string *str*)	Returns the binary equivalent of the numeric string in *str*. If the string does not represent a **decimal** value, an exception is thrown.
public static decimal Parse(string *str*, IFormatProvider *fmtpvdr*)	Returns the binary equivalent of the numeric string in *str* using the culture-specific information provided by *fmtpvdr*. If the string does not represent a **decimal** value, an exception is thrown.
public static decimal Parse(string *str*, NumberStyles *styles*)	Returns the binary equivalent of the numeric string in *str*, using the style information provided by *styles*. If the string does not represent a **decimal** value, an exception is thrown.
public static decimal Parse(string *str*, NumberStyles *styles*, IFormatProvider *fmtpvdr*)	Returns the binary equivalent of the numeric string in *str* using the style information provided by *styles* and the culture-specific format information provided by *fmtpvdr*. If the string does not represent a **decimal** value, an exception is thrown.
public static decimal Remainder(decimal *v1*, decimal *v2*)	Returns the remainder of the integer division *v1* / *v2*.
public static decimal Round(decimal *v*)	Returns the value of *v* rounded to the nearest whole number.
public static decimal Round(decimal *v*, int *decPlaces*)	Returns the value of *v* rounded to the number of decimal places specified by *decPlaces*, which must be between 0 and 28.

TABLE 21-7 Methods Defined by **Decimal** *(continued)*

Method	Meaning
public static decimal Round(decimal *v*, MidPointRounding *how*)	Returns the value of *v* rounded to the nearest whole number using the rounding mode specified by *how.* The rounding mode applies only to those conditions in which *v* is at the midpoint between two whole numbers.
public static decimal Round(decimal *v*, int *decPlaces*, MidPointRounding *how*)	Returns the value of *v* rounded to the number of decimal places specified by *decPlaces* (which must be between 0 and 28), using the rounding mode specified by *how*. The rounding mode applies only to those conditions in which *v* is at the midpoint between two rounded values.
public static decimal Subtract(decimal *v1*, decimal *v2*)	Returns *v1 – v2.*
public static byte ToByte(decimal *v*)	Returns the **byte** equivalent of *v.* Any fractional component is truncated. An **OverflowException** occurs if *v* is not within the range of a **byte**.
public static double ToDouble(decimal *v*)	Returns the **double** equivalent of *v.* A loss of precision may occur because **double** has fewer significant digits than does **decimal**.
public static short ToInt16(decimal *v*)	Returns the **short** equivalent of *v.* Any fractional component is truncated. An **OverflowException** occurs if *v* is not within the range of a **short**.
public static int ToInt32(decimal *v*)	Returns the **int** equivalent of *v.* Any fractional component is truncated. An **OverflowException** occurs if *v* is not within the range of an **int**.
public static long ToInt64(decimal *v*)	Returns the **long** equivalent of *v.* Any fractional component is truncated. An **OverflowException** occurs if *v* is not within the range of a **long**.
public static long ToOACurrency(decimal *v*)	Converts *v* into the equivalent OLE Automation currency value and returns the result.
public static sbyte ToSByte(decimal *v*)	Returns the **sbyte** equivalent of *v.* Any fractional component is truncated. An **OverflowException** occurs if *v* is not within the range of an **sbyte**.
public static float ToSingle(decimal *v*)	Returns the **float** equivalent of *v.* A loss of precision may occur because **float** has fewer significant digits than does **decimal**.
public override string ToString( )	Returns the string representation of the value of the invoking object in the default format.
public string ToString(string *format*)	Returns the string representation of the value of the invoking object as specified by the format string passed in *format.*

TABLE 21-7 Methods Defined by **Decimal** *(continued)*

Method	Meaning
public string ToString(IFormatProvider *fmtpvdr*)	Returns the string representation of the value of the invoking object using the culture-specific information specified in *fmtpvdr.*
public string ToString(string *format*, IFormatProvider *fmtpvdr*)	Returns the string representation of the value of the invoking object using the culture-specific information specified in *fmtpvdr* and the format specified by *format.*
public static ushort ToUInt16(decimal *v*)	Returns the **ushort** equivalent of *v.* Any fractional component is truncated. An **OverflowException** occurs if *v* is not within the range of a **ushort**.
public static uint ToUInt32(decimal *v*)	Returns the **uint** equivalent of *v.* Any fractional component is truncated. An **OverflowException** occurs if *v* is not within the range of a **uint**.
public static ulong ToUInt64(decimal *v*)	Returns the **ulong** equivalent of *v.* Any fractional component is truncated. An **OverflowException** occurs if *v* is not within the range of a **ulong**.
public static decimal Truncate(decimal *v*)	Returns the whole-number portion of *v.* Thus, it truncates any fractional digits.
public static bool TryParse(string *str*, out decimal *val*)	Attempts to convert the numeric string in *str* into a **decimal** value. If successful, the value is stored in *val* and true is returned. If no conversion takes place, false is returned. This differs from **Parse( )**, which throws an exception on failure.
public static bool TryParse(string *str*, NumberStyles *styles*, IFormatProvider *fmtpvdr*, out decimal *val*)	Attempts to convert the numeric string in *str* into a **decimal** value using the style information provided by *styles* and the culture-specific format information provided by *fmtpvdr.* If successful, the value is stored in *val* and true is returned. If no conversion takes place, false is returned. This differs from **Parse( )**, which throws an exception on failure.

TABLE 21-7 Methods Defined by **Decimal** *(continued)*

Char

The structure corresponding to the **char** type is **Char**. It is quite useful because it supplies a large number of methods that allow you to process and categorize characters. For example, you can convert a lowercase character to uppercase by calling **ToUpper( )**. You can determine if a character is a digit by calling **IsDigit( )**.

The methods defined by **Char** are shown in Table 21-9. Notice that several, such as **ConvertFromUtf32( )** and **ConvertToUtf32( )**, give you the ability to work with both UTF-16 and UTF-32 Unicode characters. In the past, all Unicode characters could be represented by 16 bits, which is the size of a **char**. However, a few years ago the Unicode character set was expanded and more than 16 bits are required. Each Unicode character is represented by a *code point*. The way that a code point is encoded depends on the Unicode Transformation Format (UTF) being used. In UTF-16, the most common code points require one 16-bit value, but some need two 16-bit values. When two 16-bit values are needed, two **char**

Field	Meaning
public static readonly decimal MaxValue	The largest value that a **decimal** can hold.
public static readonly decimal MinusOne	The **decimal** representation of –1.
public static readonly decimal MinValue	The smallest value that a **decimal** can hold.
public static readonly decimal One	The **decimal** representation of 1.
public static readonly decimal Zero	The **decimal** representation of 0.

TABLE 21-8 Fields Supported by **Decimal**

values are used to represent it. The first character is called the *high surrogate* and the second is called the *low surrogate.* In UTF-32, each code point uses one 32-bit value. **Char** provides the necessary conversions between UTF-16 and UTF-32.

Char defines the following fields:

```
public const char MaxValue
public const char MinValue
```

These represent the largest and smallest values that a **char** variable can hold.

Char implements the following interfaces: **IComparable**, **IComparable<char>**, **IConvertible**, and **IEquatable<char>**.

Method	Meaning
public int CompareTo(char *v*)	Compares the character in the invoking object with that of *v.* Returns zero if the characters are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
public int CompareTo(object *v*)	Compares the character in the invoking object with that of *v.* Returns zero if the characters are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
public static string ConvertFromUtf32(int *utf32Ch*)	Converts the Unicode UTF-32 code point in *utf32Ch* into a UTF-16 string and returns the result.
pubic static int ConvertToUtf32(char *highSurrogate*, char *lowSurrogate*)	Converts the high and low UTF-16 surrogates specified by *highSurrogate* and *lowSurrogate* into a UTF-32 code point. The result is returned.
pubic static int ConvertToUtf32(string str, int *idx*)	Converts the UTF-16 surrogate pair at *str*[*idx*] into its UTF-32 code point. The result is returned.
public bool Equals(char *v*)	Returns true if the value of the invoking object equals the value of *v.*
public override bool Equals(object *v*)	Returns true if the value of the invoking object equals the value of *v.*

TABLE 21-9 Methods Defined by **Char**

Method	Meaning
public override int GetHashCode( )	Returns the hash code for the invoking object.
public static double GetNumericValue(char *ch*)	Returns the numeric value of *ch* if *ch* is a digit. Otherwise, returns –1.
public static double GetNumericValue(string *str*, int *idx*)	Returns the numeric value of *str*[*idx*] if that character is a digit. Otherwise, returns –1.
public TypeCode GetTypeCode( )	Returns the **TypeCode** enumeration value for **Char**, which is **TypeCode.Char**.
public static UnicodeCategory GetUnicodeCategory(char *ch*)	Returns the **UnicodeCategory** enumeration value for *ch*. **UnicodeCategory** is an enumeration defined by **System.Globalization** that categorizes Unicode characters.
public static UnicodeCategory GetUnicodeCategory(string *str*, int *idx*)	Returns the **UnicodeCategory** enumeration value for *str*[*idx*]. **UnicodeCategory** is an enumeration defined by **System.Globalization** that categorizes Unicode characters.
public static bool IsControl(char *ch*)	Returns true if *ch* is a control character. Otherwise, returns false.
public static bool IsControl(string *str*, int *idx*)	Returns true if *str*[*idx*] is a control character. Otherwise, returns false.
public static bool IsDigit(char *ch*)	Returns true if *ch* is a digit. Otherwise, returns false.
public static bool IsDigit(string *str*, int *idx*)	Returns true if *str*[*idx*] is a digit. Otherwise, returns false.
public static bool IsHighSurrogate(char *ch*)	Returns true if *ch* is a valid UTF-32 high surrogate. Otherwise, returns false.
public static bool IsHighSurrogate(string *str*, int *idx*)	Returns true if *str*[*idx*] is a valid UTF-32 high surrogate. Otherwise, returns false.
public static bool IsLetter(char *ch*)	Returns true if *ch* is a letter of the alphabet. Otherwise, returns false.
public static bool IsLetter(string *str*, int *idx*)	Returns true if *str*[*idx*] is a letter of the alphabet. Otherwise, returns false.
public static bool IsLetterOrDigit(char *ch*)	Returns true if *ch* is either a letter of the alphabet or a digit. Otherwise, returns false.
public static bool IsLetterOrDigit(string *str*, int *idx*)	Returns true if *str*[*idx*] is either a letter of the alphabet or a digit. Otherwise, returns false.
public static bool IsLower(char *ch*)	Returns true if *ch* is a lowercase letter of the alphabet. Otherwise, returns false.
public static bool IsLower(string *str*, int *idx*)	Returns true if *str*[*idx*] is a lowercase letter of the alphabet. Otherwise, returns false.

Table 21-9 Methods Defined by **Char** *(continued)*

Method	Meaning
public static bool IsLowSurrogate(char *ch*)	Returns true if *ch* is a valid UTF-32 low surrogate. Otherwise, returns false.
public static bool IsLowSurrogate(string *str*, int *idx*)	Returns true if *str*[*idx*] is a valid UTF-32 low surrogate. Otherwise, returns false.
public static bool IsNumber(char *ch*)	Returns true if *ch* is a number. Otherwise, returns false.
public static bool IsNumber(string *str*, int *idx*)	Returns true if *str*[*idx*] is a number. Otherwise, returns false.
public static bool IsPunctuation(char *ch*)	Returns true if *ch* is a punctuation character. Otherwise, returns false.
public static bool IsPunctuation(string *str*, int *idx*)	Returns true if *str*[*idx*] is a punctuation character. Otherwise, returns false.
public static bool IsSeparator(char *ch*)	Returns true if *ch* is a separator character, such as a space. Otherwise, returns false.
public static bool IsSeparator(string *str*, int *idx*)	Returns true if *str*[*idx*] is a separator character, such as a space. Otherwise, returns false.
public static bool IsSurrogate(char *ch*)	Returns true if *ch* is a Unicode surrogate character. Otherwise, returns false.
public static bool IsSurrogate(string *str*, int *idx*)	Returns true if *str*[*idx*] is a Unicode surrogate character. Otherwise, returns false.
public static bool IsSurrogatePair(char *high*, char *low*)	Returns true if *high* and *low* form a valid surrogate pair. Otherwise, returns false.
public static bool IsSurrogatePair(string *str*, int *idx*)	Returns true if the two consecutive characters starting at *idx* within *str* form a valid surrogate pair. Otherwise, returns false.
public static bool IsSymbol(char *ch*)	Returns true if *ch* is a symbolic character, such as the currency symbol. Otherwise, returns false.
public static bool IsSymbol(string *str*, int *idx*)	Returns true if *str*[*idx*] is a symbolic character, such as the currency symbol. Otherwise, returns false.
public static bool IsUpper(char *ch*)	Returns true if *ch* is an uppercase letter. Otherwise, returns false.
public static bool IsUpper(string *str*, int *idx*)	Returns true if *str*[*idx*] is an uppercase letter. Otherwise, returns false.
public static bool IsWhiteSpace(char *ch*)	Returns true if *ch* is a whitespace character, such as a space or tab. Otherwise, returns false.
public static bool IsWhiteSpace(string *str*, int *idx*)	Returns true if *str*[*idx*] is a whitespace character, such as a space or tab. Otherwise, returns false.

Table 21-9 Methods Defined by **Char** *(continued)*

Method	Meaning
public static char Parse(string *str*)	Returns the **char** equivalent of the character in *str*. If *str* contains more than one character, a **FormatException** is thrown.
public static char ToLower(char *ch*)	Returns the lowercase equivalent of *ch* if *ch* is an uppercase letter. Otherwise, *ch* is returned unchanged.
public static char ToLower(char *ch*, CultureInfo *c*)	Returns the lowercase equivalent of *ch* if *ch* is an uppercase letter. Otherwise, *ch* is returned unchanged. The conversion is handled in accordance with the specified cultural information. **CultureInfo** is a class defined in **System.Globalization**.
public static char ToLowerInvariant(char *ch*)	Returns the lowercase version of *ch* independently of the cultural settings.
public override string ToString( )	Returns the string representation of the value of the invoking **Char**.
public static string ToString(char *ch*)	Returns the string representation of *ch*.
public string ToString(IFormatProvider *fmtpvdr*)	Returns the string representation of the invoking **Char** using the specified cultural information.
public static char ToUpper(char *ch*)	Returns the uppercase equivalent of *ch* if *ch* is a lowercase letter. Otherwise, *ch* is returned unchanged.
public static char ToUpper(char *ch*, CultureInfo *c*)	Returns the uppercase equivalent of *ch* if *ch* is a lowercase letter. Otherwise, *ch* is returned unchanged. The conversion is handled in accordance with the specified cultural information. **CultureInfo** is a class defined in **System.Globalization**.
public static char ToUpperInvariant(char *ch*)	Returns the uppercase version of *ch* independently of the cultural settings.
public static bool TryParse(string *str*, out char *ch*)	Attempts to convert the character in *str* into its **char** equivalent. If successful, the value is stored in *ch* and true is returned. If *str* contains more than one character, false is returned. This differs from **Parse( )**, which throws an exception on failure.

TABLE 21-9 Methods Defined by **Char** *(continued)*

Here is a program that demonstrates several of the methods defined by **Char**:

```
// Demonstrate several Char methods.

using System;

class CharDemo {
```

```
  static void Main() {
    string str = "This is a test. $23";
    int i;

    for(i=0; i < str.Length; i++) {
      Console.Write(str[i] + " is");
      if(Char.IsDigit(str[i]))
        Console.Write(" digit");
      if(Char.IsLetter(str[i]))
        Console.Write(" letter");
      if(Char.IsLower(str[i]))
        Console.Write(" lowercase");
      if(Char.IsUpper(str[i]))
        Console.Write(" uppercase");
      if(Char.IsSymbol(str[i]))
        Console.Write(" symbol");
      if(Char.IsSeparator(str[i]))
        Console.Write(" separator");
      if(Char.IsWhiteSpace(str[i]))
        Console.Write(" whitespace");
      if(Char.IsPunctuation(str[i]))
        Console.Write(" punctuation");

      Console.WriteLine();
    }

    Console.WriteLine("Original: " + str);

    // Convert to uppercase.
    string newstr = "";
    for(i=0; i < str.Length; i++)
      newstr += Char.ToUpper(str[i]);

    Console.WriteLine("Uppercased: " + newstr);

  }
}
```

The output is shown here:

```
T is letter uppercase
h is letter lowercase
i is letter lowercase
s is letter lowercase
  is separator whitespace
i is letter lowercase
s is letter lowercase
  is separator whitespace
a is letter lowercase
  is separator whitespace
t is letter lowercase
e is letter lowercase
s is letter lowercase
t is letter lowercase
. is punctuation
```

```
  is separator whitespace
$ is symbol
2 is digit
3 is digit
Original: This is a test. $23
Uppercased: THIS IS A TEST. $23
```

The Boolean Structure

The **Boolean** structure supports the **bool** data type. The methods defined by **Boolean** are shown in Table 21-10. It defines these fields:

```
public static readonly string FalseString
public static readonly string TrueString
```

These contain the human-readable forms of **true** and **false**. For example, if you output **FalseString** using a call to **WriteLine( )**, the string "False" is displayed.

Boolean implements the following interfaces: **IComparable**, **IComparable<bool>**, **IConvertible**, and **IEquatable<bool>**.

Method	Meaning
public int CompareTo(bool *v*)	Compares the value of the invoking object with that of *v*. Returns zero if the values are equal. Returns a negative value if the invoking object is false and *v* is true. Returns a positive value if the invoking object is true and *v* is false.
public int CompareTo(object *v*)	Compares the value of the invoking object with that of *v*. Returns zero if the values are equal. Returns a negative value if the invoking object is false and *v* is true. Returns a positive value if the invoking object is true and *v* is false.
public bool Equals(bool *v*)	Returns true if the value of the invoking object equals the value of *v*.
public override bool Equals(object *v*)	Returns true if the value of the invoking object equals the value of *v*.
public override int GetHashCode( )	Returns the hash code for the invoking object.
public TypeCode GetTypeCode( )	Returns the **TypeCode** enumeration value for **Boolean**, which is **TypeCode.Boolean**.
public static bool Parse(string *str*)	Returns the **bool** equivalent of the string in *str*. If the string is neither **Boolean.TrueString** nor **Boolean.FalseString**, a **FormatException** is thrown. However, case differences are ignored.
public override string ToString( )	Returns the string representation of the value of the invoking object, which will be either **TrueString** or **FalseString**.
public string ToString(IFormatProvider *fmtpvdr*)	Returns the string representation of the value of the invoking object, which will be either **TrueString** or **FalseString**. The *fmtpvdr* parameter is ignored.

Table 21-10 Methods Defined by **Boolean**

Method	Meaning
public static bool TryParse(string *str*, out bool *b*)	Attempts to convert the character in *str* into its **bool** equivalent. If successful, the value is stored in *b* and true is returned. If the string is neither **Boolean.TrueString** nor **Boolean.FalseString**, false is returned. (Case differences are ignored.) This differs from **Parse( )**, which throws an exception on failure.

Table 21-10 Methods Defined by **Boolean** *(continued)*

The Array Class

One very useful class in **System** is **Array**. **Array** is a base class for all arrays in C#. Thus, its methods can be applied to arrays of any of the built-in types or to arrays of types that you create. **Array** defines the properties shown in Table 21-11. It defines the methods shown in Table 21-12.

Array implements the following interfaces: **ICloneable**, **ICollection**, **IEnumerable**, and **IList**. **ICollection**, **IEnumerable**, and **IList** are defined in the **System.Collections** namespace and are described in Chapter 24.

Several methods use a parameter of type **IComparer** or **IComparer<T>**. The **IComparer** interface is in **System.Collections**. It defines a method called **Compare( )**, which compares the values of two objects. It is shown here:

int Compare(object *v1*, object *v2*)

It returns greater than zero if *v1* is greater than *v2*, less than zero if *v1* is less than *v2*, and zero if the two values are equal.

IComparer<T> is in **System.Collections.Generic**. It defines a generic form of **Compare( )**, which is shown here:

int Compare(T *v1*, T *v2*)

It works the same as its non-generic relative: returning greater than zero if *v1* is greater than *v2*, less than zero if *v1* is less than *v2*, and zero if the two values are equal. The advantage to **IComparer<T>** is type safety, because the type of data being operated upon is explicitly specified. Thus, no casts from **object** are required.

The next few sections demonstrate several commonly used array operations.

Property	Meaning
public bool IsFixedSize { get; }	A read-only property that is true if the array is of fixed size and false if the array is dynamic. This value is true for arrays.
public bool IsReadOnly { get; }	A read-only property that is true if the **Array** object is read-only and false if it is not. This value is true for arrays.
public bool IsSynchronized { get; }	A read-only property that is true if the array is safe for use in a multithreaded environment and false if it is not. This value is true for arrays.

Table 21-11 Properties Defined by **Array**

Property	Meaning
public int Length { get; }	An **int** read-only property that contains the number of elements in the array.
public long LongLength { get; }	A **long** read-only property that contains the number of elements in the array.
public int Rank { get; }	A read-only property that contains the number of dimensions in the array.
public object SyncRoot { get; }	A read-only property that contains the object that synchronizes access to the array.

TABLE 21-11 Properties Defined by **Array** *(continued)*

Method	Meaning
public static ReadOnlyCollection<T> AsReadOnly<T>(T[ ] *a*)	Returns a read-only collection that wraps the array specified by *a*.
public static int BinarySearch(Array *a*, object *v*)	Searches the array specified by *a* for the value specified by *v*. Returns the index of the first match. If *v* is not found, returns a negative value. The array must be sorted and one-dimensional.
public static int BinarySearch<T>(T[ ] *a*, T *v*)	Searches the array specified by *a* for the value specified by *v*. Returns the index of the first match. If *v* is not found, returns a negative value. The array must be sorted and one-dimensional.
public static int BinarySearch(Array *a*, object *v*, IComparer *comp*)	Searches the array specified by *a* for the value specified by *v*, using the comparison method specified by *comp*. Returns the index of the first match. If *v* is not found, returns a negative value. The array must be sorted and one-dimensional.
public static int BinarySearch<T>(T[ ] *a*, T *v*, IComparer<T> *comp*)	Searches the array specified by *a* for the value specified by *v*, using the comparison method specified by *comp*. Returns the index of the first match. If *v* is not found, returns a negative value. The array must be sorted and one-dimensional.
public static int BinarySearch(Array *a*, int *start*, int *count*, object *v*)	Searches a portion of the array specified by *a* for the value specified by *v*. The search begins at the index specified by *start* and is restricted to *count* elements. Returns the index of the first match. If *v* is not found, returns a negative value. The array must be sorted and one-dimensional.
public static int BinarySearch<T>(T[ ] *a*, int *start*, int *count*, T *v*)	Searches a portion of the array specified by *a* for the value specified by *v*. The search begins at the index specified by *start* and is restricted to *count* elements. Returns the index of the first match. If *v* is not found, returns a negative value. The array must be sorted and one-dimensional.
public static int BinarySearch(Array *a*, int *start*, int *count*, object *v*, IComparer *comp*)	Searches a portion of the array specified by *a* for the value specified by *v*, using the comparison method specified by *comp*. The search begins at the index specified by *start* and is restricted to *count* elements. Returns the index of the first match. If *v* is not found, returns a negative value. The array must be sorted and one-dimensional.
public static int BinarySearch<T>(T [ ] *a*, int *start*, int *count*, T *v*, IComparer<T> *comp*)	Searches a portion of the array specified by *a* for the value specified by *v*, using the comparison method specified by *comp*. The search begins at the index specified by *start* and is restricted to *count* elements. Returns the index of the first match. If *v* is not found, returns a negative value. The array must be sorted and one-dimensional.

TABLE 21-12 Methods Defined by **Array**

Method	Meaning
public static void Clear(Array *a*, int *start*, int *count*)	Sets the specified elements of *a* to zero, **null**, or **false**, depending on whether the element type is a value type, a reference type, or Boolean. The elements to be zeroed begin at the index specified by *start* and run for *count* elements.
public object Clone( )	Returns a copy of the invoking array. The copy refers to the same elements as does the original. This is called a "shallow copy." Thus, changes to the elements affect both arrays since they both use the same elements.
public static void ConstrainedCopy(Array *source*, int *srcIdx*, Array *dest*, int *destIdx*, int *count*)	Copies *count* elements from *source* (beginning at *srcIdx*) to *dest* (beginning at *destIdx*). If both arrays are reference types, then **ConstrainedCopy( )** makes a "shallow copy," which means that both arrays will refer to the same elements. If an error occurs during the copy, *dest* is unchanged.
public static TTo[ ] ConvertAll<TFrom, TTo>(TFrom[ ] *a*, Converter<TFrom, TTo> *conv*)	Converts *a* from type **TFrom** to **TTo** and returns the resulting array. The original array is unaffected. The conversion is performed by the specified converter.
public static void Copy(Array *source*, Array *dest*, int *count*)	Beginning at the start of each array, copies *count* elements from *source* to *dest*. When both arrays are reference types, then **Copy( )** makes a "shallow copy," which means that both arrays will refer to the same elements. If an error occurs during the copy, *dest* is undefined.
public static void Copy(Array *source*, Array *dest*, long *count*)	Beginning at the start of each array, copies *count* elements from *source* to *dest*. When both arrays are reference types, then **Copy( )** makes a "shallow copy," which means that both arrays will refer to the same elements. If an error occurs during the copy, *dest* is undefined.
public static void Copy(Array *source*, int *srcStart*, Array *dest*, int *destStart,* int *count*)	Copies *count* elements from *source*[*srcStart*] to *dest*[*destStart*]. When both arrays are reference types, then **Copy( )** makes a "shallow copy," which means that both arrays will refer to the same elements. If an error occurs during the copy, *dest* is undefined.
public static void Copy(Array *source*, long *srcStart*, Array *dest*, long *destStart,* long *count*)	Copies *count* elements from *source*[*srcStart*] to *dest*[*destStart*]. When both arrays are reference types, then **Copy( )** makes a "shallow copy," which means that both arrays will refer to the same elements. If an error occurs during the copy, *dest* is undefined.
public void CopyTo(Array *dest*, int *start*)	Copies the elements of the invoking array to *dest,* beginning at *dest*[*start*].
public void CopyTo(Array *dest*, long *start*)	Copies the elements of the invoking array to *dest,* beginning at *dest*[*start*].
public static Array CreateInstance(Type *t*, int *size*)	Returns a reference to a one-dimensional array that contains *size* elements of type *t*.
public static Array CreateInstance(Type *t*, int *size1*, int *size2*)	Returns a reference to a *size1*-by-*size2* two-dimensional array. Each element is of type *t*.
public static Array CreateInstance(Type *t*, int *size1*, int *size2*, int *size3*)	Returns a reference to a *size1*-by-*size2*-by-*size3* three-dimensional array. Each element is of type *t*.
public static Array CreateInstance(Type *t*, int[ ] *sizes*)	Returns a reference to a multi-dimensional array that has the dimensions specified in *sizes*. Each element is of type *t*.
public static Array CreateInstance(Type *t*, long[ ] *sizes*)	Returns a reference to a multi-dimensional array that has the dimensions specified in *sizes*. Each element is of type *t*.
public static Array CreateInstance(Type *t*, int[ ] *sizes*, int[ ] *startIndexes*)	Returns a reference to a multi-dimensional array that has the dimensions specified in *sizes*. Each element is of type *t*. The starting index of each dimension is specified in *startIndexes*. Thus, it is possible to create arrays that begin at some index other than zero.

TABLE 21-12 Methods Defined by **Array** *(continued)*

Method	Meaning
public override bool Equals(object *v*)	Returns true if the value of the invoking object equals the value of *v*.
public static bool Exists<T>(T[ ] a, Predicate<T> *pred*)	Returns true if *a* contains at least one element that satisfies the predicate specified by *pred*. Returns false if no elements satisfy *pred*.
public static T Find<T>(T[ ] *a*, Predicate<T> *pred*)	Returns the first element in *a* that satisfies the predicate specified by *pred*. If no element satisfies *pred,* then **default(T)** is returned.
public static T[ ] FindAll<T>(T[ ] *a*, Predicate<T> *pred*)	Returns an array that contains all elements in *a* that satisfy the predicate specified by *pred*. If no element satisfies *pred,* then a zero-length array is returned.
public static int FindIndex<T>(T[ ] *a*, Predicate<T> *pred*)	Returns the index of the first element in *a* that satisfies the predicate specified by *pred*. If no element satisfies *pred,* –1 is returned.
public static int FindIndex<T>(T[ ] *a*, int *start*, Predicate<T> *pred*)	Returns the index of the first element in *a* that satisfies the predicate specified by *pred*. The search begins at *a*[*start*]. If no element satisfies *pred,* –1 is returned.
public static int FindIndex<T>(T[ ] *a*, int *start*, int *count*, Predicate<T> *pred*)	Returns the index of the first element in *a* that satisfies the predicate specified by *pred*. The search begins at *a*[*start*] and runs for *count* elements. If no element satisfies *pred,* –1 is returned.
public static T FindLast<T>(T[ ] *a*, Predicate<T> *pred*)	Returns the last element in *a* that satisfies the predicate specified by *pred*. If no element satisfies *pred,* then **default(T)** is returned.
public static int FindLastIndex<T>(T[ ] *a*, Predicate<T> *pred*)	Returns the index of the last element in *a* that satisfies the predicate specified by *pred*. If no element satisfies *pred,* –1 is returned.
public static int FindLastIndex<T>(T[ ] *a*, int *start*, Predicate<T> *pred*)	Returns the index of the last element in *a* that satisfies the predicate specified by *pred*. The search proceeds in reverse order, beginning at *a*[*start*] and stopping at *a*[0]. If no element satisfies *pred,* –1 is returned.
public static int FindLastIndex<T>(T[ ] *a*, int *start*, int *count*, Predicate<T> *pred*)	Returns the index of the last element in *a* that satisfies the predicate specified by *pred*. The search proceeds in reverse order, beginning at *a*[*start*] and running for *count* elements. If no element satisfies *pred*, –1 is returned.
public static void ForEach<T>(T[ ] a, Action<T> *act*)	Applies the method specified by *act* to each element of *a*.
public IEnumerator GetEnumerator( )	Returns an enumerator object for the array. An enumerator enables you to cycle through an array. Enumerators are described in Chapter 24.
public override int GetHashCode( )	Returns the hash code for the invoking object.
public int GetLength(int *dim*)	Returns the length of the specified dimension. The dimension is zero-based. Thus, to get the length of the first dimension, pass 0; to obtain the length of the second dimension, pass 1; and so on.
public long GetLongLength(int *dim*)	Returns the length of the specified dimension as a **long**. The dimension is zero-based. Thus, to get the length of the first dimension, pass 0; to obtain the length of the second dimension, pass 1; and so on.
public int GetLowerBound(int *dim*)	Returns the first index of the specified dimension, which is usually zero. The parameter *dim* is zero-based. Thus, to get the start index of the first dimension, pass 0; to obtain the start index of the second dimension, pass 1; and so on.
public int GetUpperBound(int *dim*)	Returns the last index of the specified dimension. The parameter *dim* is zero-based. Thus, to get the last index of the first dimension, pass 0; to obtain the last index of the second dimension, pass 1; and so on.

TABLE 21-12 Methods Defined by **Array** *(continued)*

Method	Meaning
public object GetValue(int *idx*)	Returns the value of the element at index *idx* within the invoking array. The array must be one-dimensional.
public object GetValue(long *idx*)	Returns the value of the element at index *idx* within the invoking array. The array must be one-dimensional.
public object GetValue(int *idx1*, int *idx2*)	Returns the value of the element at [*idx1*, *idx2*] within the invoking array. The array must be two-dimensional.
public object GetValue(long *idx1*, long *idx2*)	Returns the value of the element at [*idx1*, *idx2*] within the invoking array. The array must be two-dimensional.
public object GetValue(int *idx1*, int *idx2*, int *idx3*)	Returns the value of the element at [*idx1*, *idx2*, *idx3*] within the invoking array. The array must be three-dimensional.
public object GetValue(long *idx1*, long *idx2*, long *idx3*)	Returns the value of the element at [*idx1*, *idx2*, *idx3*] within the invoking array. The array must be three-dimensional.
public object GetValue(int[ ] *idxs*)	Returns the value of the element at the specified indices within the invoking array. The array must have as many dimensions as *idxs* has elements.
public object GetValue(long[ ] *idxs*)	Returns the value of the element at the specified indices within the invoking array. The array must have as many dimensions as *idxs* has elements.
public static int IndexOf(Array *a*, object *v*)	Returns the index of the first element within the one-dimensional array *a* that has the value specified by *v*. Returns –1 if the value is not found. (If the array has a lower bound other than 0, then the failure value is the lower bound –1.)
public static int IndexOf<T>(T[ ] *a*, T *v*)	Returns the index of the first element within the one-dimensional array *a* that has the value specified by *v*. Returns –1 if the value is not found.
public static int IndexOf(Array *a*, object *v*, int *start*)	Returns the index of the first element within the one-dimensional array *a* that has the value specified by *v*. The search begins at *a*[*start*]. Returns –1 if the value is not found. (If the array has a lower bound other than 0, then the failure value is the lower bound –1.)
public static int IndexOf<T>(T[ ] *a*, T *v*, int *start*)	Returns the index of the first element within the one-dimensional array *a* that has the value specified by *v*. The search begins at *a*[*start*]. Returns –1 if the value is not found.
public static int IndexOf(Array *a*, object *v*, int *start*, int *count*)	Returns the index of the first element within the one-dimensional array *a* that has the value specified by *v*. The search begins at *a*[*start*] and runs for *count* elements. Returns –1 if the value is not found within the specified range. (If the array has a lower bound other than 0, then the failure value is the lower bound –1.)
public static int IndexOf<T>(T[ ] *a*, T *v*, int *start*, int *count*)	Returns the index of the first element within the one-dimensional array *a* that has the value specified by *v*. The search begins at *a*[*start*] and runs for *count* elements. Returns –1 if the value is not found within the specified range.
public void Initialize( )	Initializes each element in the invoking array by calling the element's default constructor. This method can be used only on arrays of value types that have constructors.
public static int LastIndexOf(Array *a*, object *v*)	Returns the index of the last element within the one-dimensional array *a* that has the value specified by *v*. Returns –1 if the value is not found. (If the array has a lower bound other than 0, then the failure value is the lower bound –1.)
public static int LastIndexOf<T>(T[ ] *a*, T *v*)	Returns the index of the last element within the one-dimensional array *a* that has the value specified by *v*. Returns –1 if the value is not found.
public static int LastIndexOf(Array *a*, object *v*, int *start*)	Returns the index of the last element within a range of the one-dimensional array *a* that has the value specified by *v*. The search proceeds in reverse order, beginning at *a*[*start*] and stopping at *a*[0]. Returns –1 if the value is not found. (If the array has a lower bound other than 0, then the failure value is the lower bound –1.)

TABLE 21-12 Methods Defined by **Array** *(continued)*

Method	Meaning
public static int LastIndexOf<T>(T[ ] *a*, T *v*, int *start*)	Returns the index of the last element within a range of the one-dimensional array *a* that has the value specified by *v*. The search proceeds in reverse order, beginning at *a*[*start*] and stopping at *a*[0]. Returns –1 if the value is not found.
public static int LastIndexOf(Array *a*, object *v*, int *start*, int *count*)	Returns the index of the last element within a range of the one-dimensional array *a* that has the value specified by *v*. The search proceeds in reverse order, beginning at *a*[*start*] and running for *count* elements. Returns –1 if the value is not found within the specified range. (If the array has a lower bound other than 0, then the failure value is the lower bound –1.)
public static int LastIndexOf<T>(T[ ] *a*, T *v*, int *start*, int *count*)	Returns the index of the last element within a range of the one-dimensional array *a* that has the value specified by *v*. The search proceeds in reverse order, beginning at *a*[*start*] and running for *count* elements. Returns –1 if the value is not found within the specified range.
public static void Resize<T>(ref T[ ] *a*, int *size*)	Sets the size of *a* to *size*.
public static void Reverse(Array *a*)	Reverses the elements in *a*.
public static void Reverse(Array *a*, int *start*, int *count*)	Reverses a range of elements in *a*. The range reversed begins at *a*[*start*] and runs for *count* elements.
public void SetValue(object *v*, int *idx*)	Sets the value of the element at index *idx* within the invoking array to *v*. The array must be one-dimensional.
public void SetValue(object *v*, long *idx*)	Sets the value of the element at index *idx* within the invoking array to *v*. The array must be one-dimensional.
public void SetValue(object *v*, int *idx1*, int *idx2*)	Sets the value of the element at indices [*idx1*, *idx2*] within the invoking array to *v*. The array must be two-dimensional.
public void SetValue(object *v*, long *idx1*, long *idx2*)	Sets the value of the element at indices [*idx1*, *idx2*] within the invoking array to *v*. The array must be two-dimensional.
public void SetValue(object *v*, int *idx1*, int *idx2*, int *idx3*)	Sets the value of the element at indices [*idx1*, *idx2*, *idx3*] within the invoking array to *v*. The array must be three-dimensional.
public void SetValue(object *v*, long *idx1*, long *idx2*, long *idx3*)	Sets the value of the element at indices [*idx1*, *idx2*, *idx3*] within the invoking array to *v*. The array must be three-dimensional.
public void SetValue(object *v*, int[ ] *idxs*)	Sets the value of the element at the specified indices within the invoking array to *v*. The array must have as many dimensions as *idxs* has elements.
public void SetValue(object *v*, long[ ] *idxs*)	Sets the value of the element at the specified indices within the invoking array to *v*. The array must have as many dimensions as *idxs* has elements.
public static void Sort(Array *a*)	Sorts *a* into ascending order. The array must be one-dimensional.
public static void Sort<T>(T[ ] *a*)	Sorts *a* into ascending order. The array must be one-dimensional.
public static void Sort(Array *a*, IComparer *comp*)	Sorts *a* into ascending order using the comparison method specified by *comp*. The array must be one-dimensional.
public static void Sort<T>(T[ ] *a*, Comparison<T> *comp*)	Sorts *a* into ascending order using the comparison method specified by *comp*. The array must be one-dimensional.
public static void Sort<T>(T[ ] *a*, IComparer<T> *comp*)	Sorts *a* into ascending order using the comparison method specified by *comp*. The array must be one-dimensional.
public static void Sort(Array *k*, Array *v*)	Sorts a pair of one-dimensional arrays into ascending order. The *k* array contains the sort keys. The *v* array contains the values linked to those keys. Thus, the two arrays contain key/value pairs. After the sort, both arrays are in ascending-key order.

Table 21-12 Methods Defined by **Array** *(continued)*

Method	Meaning
public static void Sort<TK, TV>(TK[ ] *k*, TV[ ] *v*)	Sorts a pair of one-dimensional arrays into ascending order. The *k* array contains the sort keys. The *v* array contains the values linked to those keys. Thus, the two arrays contain key/value pairs. After the sort, both arrays are in ascending-key order.
public static void Sort(Array *k*, Array *v*, IComparer *comp*)	Sorts a pair of one-dimensional arrays into ascending order using the comparison method specified by *comp*. The *k* array contains the sort keys. The *v* array contains the values linked to those keys. Thus, the two arrays contain key/value pairs. After the sort, both arrays are in ascending-key order.
public static void Sort<TK, TV>(TK[ ] *k*, TV[ ] *v*, IComparer<TK> *comp*)	Sorts a pair of one-dimensional arrays into ascending order using the comparison method specified by *comp*. The *k* array contains the sort keys. The *v* array contains the values linked to those keys. Thus, the two arrays contain key/value pairs. After the sort, both arrays are in ascending-key order.
public static void Sort(Array *a*, int *start*, int *count*)	Sorts a range of *a* into ascending order. The range begins at *a*[*start*] and runs for *count* elements. The array must be one-dimensional.
public static void Sort<T>(T[ ] *a*, int *start*, int *count*)	Sorts a range of *a* into ascending order. The range begins at *a*[*start*] and runs for *count* elements. The array must be one-dimensional.
public static void Sort(Array *a*, int *start*, int *count*, IComparer *comp*)	Sorts a range of *a* into ascending order using the comparison method specified by *comp*. The range begins at *a*[*start*] and runs for *count* elements. The array must be one-dimensional.
public static void Sort<T>(T[ ] *a*, int *start*, int *count*, IComparer<T> *comp*)	Sorts a range of *a* into ascending order using the comparison method specified by *comp*. The range begins at *a*[*start*] and runs for *count* elements. The array must be one-dimensional.
public static void Sort(Array *k*, Array *v*, int *start*, int *count*)	Sorts a range within a pair of one-dimensional arrays into ascending order. Within both arrays, the range to sort begins at the index passed in *start* and runs for *count* elements. The *k* array contains the sort keys. The *v* array contains the values linked to those keys. Thus, the two arrays contain key/value pairs. After the sort, both ranges are in ascending-key order.
public static void Sort<TK, TV>(TK[ ] *k*, TK[ ] *v*, int *start*, int *count*)	Sorts a range within a pair of one-dimensional arrays into ascending order. Within both arrays, the range to sort begins at the index passed in *start* and runs for *count* elements. The *k* array contains the sort keys. The *v* array contains the values linked to those keys. Thus, the two arrays contain key/value pairs. After the sort, both ranges are in ascending-key order.
public static void Sort(Array *k*, Array *v*, int *start*, int *count*, IComparer *comp*)	Sorts a range within a pair of one-dimensional arrays into ascending order using the comparison method specified by *comp*. Within both arrays, the range to sort begins at the index passed in *start* and runs for *count* elements. The *k* array contains the sort keys. The *v* array contains the values linked to those keys. Thus, the two arrays contain key/value pairs. After the sort, both ranges are in ascending-key order.
public static void Sort<TK, TV>(TK[ ] *k*, TV *v*, int *start*, int *count*, IComparer<TK> *comp*)	Sorts a range within a pair of one-dimensional arrays into ascending order using the comparison method specified by *comp*. Within both arrays, the range to sort begins at the index passed in *start* and runs for *count* elements. The *k* array contains the sort keys. The *v* array contains the values linked to those keys. Thus, the two arrays contain key/value pairs. After the sort, both ranges are in ascending-key order.
public static bool TrueForAll<T>(T[ ] a, Predicate<T> *pred*)	Returns true if the predicate specified by *pred* is satisfied by all elements in *a*. If one or more elements fail to satisfy *pred*, then false is returned.

TABLE 21-12 Methods Defined by **Array** *(continued)*

Sorting and Searching Arrays

Often you will want to sort the contents of an array. To handle this, **Array** supports a rich complement of sorting methods. Using **Sort( )**, you can sort an entire array, a range within an array or a pair of arrays that contain corresponding key/value pairs. Once an array has been sorted, you can efficiently search it using **BinarySearch( )**. Here is a program that demonstrates the **Sort( )** and **BinarySearch( )** methods by sorting an array of **int**s:

```
// Sort an array and search for a value.

using System;

class SortDemo {
  static void Main() {
    int[] nums = { 5, 4, 6, 3, 14, 9, 8, 17, 1, 24, -1, 0 };

    // Display original order.
    Console.Write("Original order: ");
    foreach(int i in nums)
      Console.Write(i + " ");
    Console.WriteLine();

    // Sort the array.
    Array.Sort(nums);

    // Display sorted order.
    Console.Write("Sorted order:   ");
    foreach(int i in nums)
      Console.Write(i + " ");
    Console.WriteLine();


    // Search for 14.
    int idx = Array.BinarySearch(nums, 14);

    Console.WriteLine("Index of 14 is " + idx);
  }
}
```

The output is shown here:

```
Original order: 5 4 6 3 14 9 8 17 1 24 -1 0
Sorted order:   -1 0 1 3 4 5 6 8 9 14 17 24
Index of 14 is 9
```

In the preceding example, the array has an element type of **int**, which is a value type. All methods defined by **Array** are automatically available to all of the built-in value types. However, this may not be the case for arrays of object references. To sort or search an array of object references, the class type of those objects must implement either the **IComparable** or **IComparable<T>** interface. If the class does not implement one of these interfaces, a runtime exception will occur when attempting to sort or search the array. Fortunately, both **IComparable** and **IComparable<T>** are easy to implement.

IComparable defines just one method:

int CompareTo(object *obj*)

This method compares the invoking object against the value in *obj*. It returns greater than zero if the invoking object is greater than *obj*, zero if the two objects are equal, and less than zero if the invoking object is less than *obj*.

IComparable<T> is the generic version of **IComparable**. It defines the generic version of **CompareTo( )**:

int CompareTo(T *obj*)

The generic version of **CompareTo( )** works like the non-generic version. It compares the invoking object against the value in *obj*. It returns greater than zero if the invoking object is greater than *obj*, zero if the two objects are equal, and less than zero if the invoking object is less than *obj*. The advantage of using **IComparable<T>** is type safety because the type of data being operated upon is explicitly specified. There is no need to cast the object being compared from **object** into the desired type. Here is an example that illustrates sorting and searching an array of user-defined class objects:


```
// Sort and search an array of objects.

using System;

class MyClass : IComparable<MyClass> {
  public int i;

  public MyClass(int x) { i = x; }

  // Implement IComparable<MyClass>.
  public int CompareTo(MyClass v) {
    return i - v.i;
  }

  public bool Equals(MyClass v) {
    return i == v.i;
  }

}

class SortDemo {
  static void Main() {
    MyClass[] nums = new MyClass[5];

    nums[0] = new MyClass(5);
    nums[1] = new MyClass(2);
    nums[2] = new MyClass(3);
    nums[3] = new MyClass(4);
    nums[4] = new MyClass(1);

    // Display original order.
    Console.Write("Original order: ");
    foreach(MyClass o in nums)
      Console.Write(o.i + " ");
    Console.WriteLine();

    // Sort the array.
    Array.Sort(nums);
```

```
    // Display sorted order.
    Console.Write("Sorted order:   ");
    foreach(MyClass o in nums)
      Console.Write(o.i + " ");
    Console.WriteLine();

    // Search for MyClass(2).
    MyClass x = new MyClass(2);
    int idx = Array.BinarySearch(nums, x);

    Console.WriteLine("Index of MyClass(2) is " + idx);
  }
}
```

The output is shown here:

```
Original order: 5 2 3 4 1
Sorted order:   1 2 3 4 5
Index of MyClass(2) is 1
```

Reversing an Array

Sometimes it is useful to reverse the contents of an array. For example, you might want to change an array that has been sorted in ascending order into one sorted in descending order. Reversing an array is easy: Simply call **Reverse( )**. Using **Reverse( )**, you can reverse all or part of an array. The following program demonstrates the process:

```
// Reverse an array.

using System;

class ReverseDemo {
  static void Main() {
    int[] nums = { 1, 2, 3, 4, 5 };

    // Display original order.
    Console.Write("Original order: ");
    foreach(int i in nums)
      Console.Write(i + " ");
    Console.WriteLine();

    // Reverse the entire array.
    Array.Reverse(nums);

    // Display reversed order.
    Console.Write("Reversed order: ");
    foreach(int i in nums)
      Console.Write(i + " ");
    Console.WriteLine();

    // Reverse a range.
    Array.Reverse(nums, 1, 3);

    // Display reversed order.
    Console.Write("Range reversed: ");
    foreach(int i in nums)
```

```
      Console.Write(i + " ");
    Console.WriteLine();
  }
}
```

The output is shown here:

```
Original order: 1 2 3 4 5
Reversed order: 5 4 3 2 1
Range reversed: 5 2 3 4 1
```

Copying an Array

Copying all or part of one array to another is another common array operation. To copy an array, use **Copy( )**. **Copy( )** can put elements at the start of the destination array or in the middle, depending upon which version of **Copy( )** you use. **Copy( )** is demonstrated by the following program:

```
// Copy an array.

using System;

class CopyDemo {
  static void Main() {
    int[] source = { 1, 2, 3, 4, 5 };
    int[] target = { 11, 12, 13, 14, 15 };
    int[] source2 = { -1, -2, -3, -4, -5 };

    // Display source.
    Console.Write("source: ");
    foreach(int i in source)
      Console.Write(i + " ");
    Console.WriteLine();

    // Display original target.
    Console.Write("Original contents of target: ");
    foreach(int i in target)
      Console.Write(i + " ");
    Console.WriteLine();

    // Copy the entire array.
    Array.Copy(source, target, source.Length);

    // Display copy.
    Console.Write("target after copy:  ");
    foreach(int i in target)
      Console.Write(i + " ");
    Console.WriteLine();

    // Copy into middle of target.
    Array.Copy(source2, 2, target, 3, 2);

    // Display copy.
    Console.Write("target after copy:  ");
    foreach(int i in target)
```

```
      Console.Write(i + " ");
    Console.WriteLine();
  }
}
```

The output is shown here:

```
source: 1 2 3 4 5
Original contents of target: 11 12 13 14 15
target after copy:  1 2 3 4 5
target after copy:  1 2 3 -3 -4
```

Using a Predicate

A *predicate* is a delegate of type **System.Predicate** that returns either true or false, based upon some condition. It is declared as shown here:

public delegate bool Predicate<T> (T *obj*)

The object to be tested against the condition is passed in *obj*. If *obj* satisfies that condition, the predicate must return true. Otherwise, it must return false. Predicates are used by several methods in **Array**, including **Exists( )**, **Find( )**, **FindIndex( )**, and **FindAll( )**.

The following program demonstrates using a predicate to determine if an array of integers contains a negative value. If a negative value is found, the program then obtains the first negative value in the array. To accomplish this, the program uses **Exists( )** and **Find( )**.

```
// Demonstrate Predicate delegate.

using System;

class PredDemo {

  // A predicate method.
  // It returns true if v is negative.
  static bool IsNeg(int v) {
    if(v < 0) return true;
    return false;
  }

  static void Main() {
    int[] nums = { 1, 4, -1, 5, -9 };

    Console.Write("Contents of nums: ");
    foreach(int i in nums)
      Console.Write(i + " ");
    Console.WriteLine();

    // First see if nums contains a negative value.
    if(Array.Exists(nums, PredDemo.IsNeg)) {
      Console.WriteLine("nums contains a negative value.");

      // Now, find first negative value.
      int x = Array.Find(nums, PredDemo.IsNeg);
      Console.WriteLine("First negative value is : " + x);
    }
```

```
    else
      Console.WriteLine("nums contains no negative values.");
  }
}
```

The output is shown here:

```
Contents of nums: 1 4 -1 5 -9
nums contains a negative value.
First negative value is : -1
```

In the program, the method passed to **Exists( )** and **Find( )** for the predicate is **IsNeg( )**. Notice that **IsNeg( )** is declared like this:

```
static bool IsNeg(int v) {
```

The methods **Exists( )** and **Find( )** will automatically pass the elements of the array (in sequence) to **v**. Thus, each time **IsNeg( )** is called, **v** will contain the next element in the array.

Using an Action

The **Action** delegate is used by **Array.ForEach( )** to perform an action on each element of an array. There are various forms of **Action**, each taking a different number of type parameters. The one used here is

public delegate void Action<T> (T *obj*)

The object to be acted upon is passed in *obj*. When used with **ForEach( )**, each element of the array is passed to *obj* in turn. Thus, through the use of **ForEach( )** and **Action**, you can, in a single statement, perform an operation over an entire array.

The following program demonstrates both **ForEach( )** and **Action**. It first creates an array of **MyClass** objects, then uses the method **Show( )** to display the values. Next, it uses **Neg( )** to negate the values. Finally, it uses **Show( )** again to display the negated values. These operations all occur through calls to **ForEach( )**.

```
// Demonstrate an Action.

using System;

class MyClass {
  public int i;

  public MyClass(int x) { i = x; }
}

class ActionDemo {

  // An Action method.
  // It displays the value it is passed.
  static void Show(MyClass o) {
    Console.Write(o.i + " ");
  }

  // Another Action method.
  // It negates the value it is passed.
```

```
  static void Neg(MyClass o) {
    o.i = -o.i;
  }

  static void Main() {
    MyClass[] nums = new MyClass[5];

    nums[0] = new MyClass(5);
    nums[1] = new MyClass(2);
    nums[2] = new MyClass(3);
    nums[3] = new MyClass(4);
    nums[4] = new MyClass(1);

    Console.Write("Contents of nums: ");

    // Use action to show the values.
    Array.ForEach(nums, ActionDemo.Show);

    Console.WriteLine();

    // Use action to negate the values.
    Array.ForEach(nums, ActionDemo.Neg);

    Console.Write("Contents of nums negated: ");

    // Use action to show the values again.
    Array.ForEach(nums, ActionDemo.Show);

    Console.WriteLine();
  }
}
```

The output is shown here:

```
Contents of nums: 5 2 3 4 1
Contents of nums negated: -5 -2 -3 -4 -1
```

BitConverter

In programming one often needs to convert a built-in data type into an array of bytes. For example, some hardware device might require an integer value, but that value must be sent one byte at a time. The reverse situation also frequently occurs. Sometimes data will be received as an ordered sequence of bytes that needs to be converted into one of the built-in types. For example, a device might output integers, sent as a stream of bytes. Whatever your conversion needs, .NET provides the **BitConverter** class to meet them.

BitConverter is **static** class. It contains the methods shown in Table 21-13. It defines the following field:

```
public static readonly bool IsLittleEndian
```

This field is **true** if the current environment stores a word with the least significant byte first and the most significant byte last. This is called "little-endian" format. **IsLittleEndian** is **false** if the current environment stores a word with the most significant byte first and the least significant byte last. This is called "big-endian" format. Intel Pentium–based machines use little-endian format.

Method	Meaning
public static long DoubleToInt64Bits(double *v*)	Converts *v* into a **long** integer and returns the result.
public static byte[ ] GetBytes(bool *v*)	Converts *v* into a 1-byte array and returns the result.
public static byte[ ] GetBytes(char *v*)	Converts *v* into a 2-byte array and returns the result.
public static byte[ ] GetBytes(double *v*)	Converts *v* into an 8-byte array and returns the result.
public static byte[ ] GetBytes(float *v*)	Converts *v* into a 4-byte array and returns the result.
public static byte[ ] GetBytes(int *v*)	Converts *v* into a 4-byte array and returns the result.
public static byte[ ] GetBytes(long *v*)	Converts *v* into an 8-byte array and returns the result.
public static byte[ ] GetBytes(short *v*)	Converts *v* into a 2-byte array and returns the result.
public static byte[ ] GetBytes(uint *v*)	Converts *v* into a 4-byte array and returns the result.
public static byte[ ] GetBytes(ulong *v*)	Converts *v* into an 8-byte array and returns the result.
public static byte[ ] GetBytes(ushort *v*)	Converts *v* into a 2-byte array and returns the result.
public static double Int64BitsToDouble(long *v*)	Converts *v* into a **double** value and returns the result.
public static bool ToBoolean(byte[ ] *a*, int *idx*)	Converts the byte at *a*[*idx*] into its **bool** equivalent and returns the result. A non-zero value is converted to true; zero is converted to false.
public static char ToChar(byte[ ] *a*, int *start*)	Converts two bytes starting at *a*[*start*] into its **char** equivalent and returns the result.
public static double ToDouble(byte[ ] *a*, int *start*)	Converts eight bytes starting at *a*[*start*] into its **double** equivalent and returns the result.
public static short ToInt16(byte[ ] *a*, int *start*)	Converts two bytes starting at *a*[*start*] into its **short** equivalent and returns the result.
public static int ToInt32(byte[ ] *a*, int *start*)	Converts four bytes starting at *a*[*start*] into its **int** equivalent and returns the result.
public static long ToInt64(byte[ ] *a*, int *start*)	Converts eight bytes starting at *a*[*start*] into its **long** equivalent and returns the result.
public static float ToSingle(byte[ ] *a*, int *start*)	Converts four bytes starting at *a*[*start*] into its **float** equivalent and returns the result.

Table 21-13 Methods Defined by **BitConverter**

Method	Meaning
public static string ToString(byte[ ] *a*)	Converts the bytes in *a* into a string. The string contains the hexadecimal values associated with the bytes, separated by hyphens.
public static string ToString(byte[ ] *a*, int *start*)	Converts the bytes in *a,* beginning at *a*[*start*], into a string. The string contains the hexadecimal values associated with the bytes, separated by hyphens.
public static string ToString(byte[ ] *a*, int *start*, int *count*)	Converts the bytes in *a,* beginning at *a*[*start*] and running for *count* bytes, into a string. The string contains the hexadecimal values associated with the bytes, separated by hyphens.
public static ushort ToUInt16(byte[ ] *a*, int *start*)	Converts two bytes starting at *a*[*start*] into its **ushort** equivalent and returns the result.
public static uint ToUInt32(byte[ ] *a*, int *start*)	Converts four bytes starting at *a*[*start*] into its **uint** equivalent and returns the result.
public static ulong ToUInt64(byte[ ] *a*, int *start*)	Converts eight bytes starting at *a*[*start*] into its **ulong** equivalent and returns the result.

TABLE 21-13 Methods Defined by **BitConverter** *(continued)*

Generating Random Numbers with Random

To generate a sequence of pseudorandom numbers, you will use the **Random** class. Sequences of random numbers are useful in a variety of situations, including simulations and modeling. The starting point of the sequence is determined by a *seed* value, which can be automatically provided by **Random** or explicitly specified.

Random defines these two constructors:

public Random( )
public Random(int *seed*)

The first version creates a **Random** object that uses the system time to compute the seed value. The second uses the value of *seed* as the seed value.

Random defines the methods shown in Table 21-14.

Here is a program that demonstrates **Random** by creating a pair of computerized dice:

```
// An automated pair of dice.

using System;

class RandDice {
  static void Main() {
    Random ran = new Random();

    Console.Write(ran.Next(1, 7) + " ");
    Console.WriteLine(ran.Next(1, 7));
  }
}
```

Method	Meaning
public virtual int Next( )	Returns the next random integer, which will be between 0 and **Int32.MaxValue**–1, inclusive.
public virtual int Next(int *upperBound*)	Returns the next random integer that is between 0 and *upperBound*–1, inclusive.
public virtual int Next(int *lowerBound*, int *upperBound*)	Returns the next random integer that is between *lowerBound* and *upperBound*–1, inclusive.
public virtual void NextBytes(byte[ ] *buf*)	Fills *buf* with a sequence of random integers. Each byte in the array will be between 0 and **Byte.MaxValue–1**, inclusive.
public virtual double NextDouble( )	Returns the next random value from the sequence represented as a floating-point number that is greater than or equal to 0.0 and less than 1.0.
protected virtual double Sample( )	Returns the next random value from the sequence represented as a floating-point number that is greater than or equal to 0.0 and less than 1.0. To create a skewed or specialized distribution, override this method in a derived class.

TABLE 21-14 Methods Defined by **Random**

Here are three sample runs:

```
5 2
4 4
1 6
```

The program works by first creating a **Random** object. Then it requests the two random values, each between 1 and 6, inclusive.

Memory Management and the GC Class

The **GC** class encapsulates the garbage-collection facility. The methods defined by **GC** are shown in Table 21-15. It defines the read-only property shown here:

```
public static int MaxGeneration { get; }
```

MaxGeneration contains the maximum generation number available to the system. A generation number indicates the age of an allocation. Newer allocations have a lower number than older ones. Generation numbers help improve the efficiency of the garbage collector.

For most applications, you will not use any of the capabilities of **GC**. However, in specialized cases, they can be very useful. For example, you might want to use **Collect( )** to force garbage collection to occur at a time of your choosing. Normally, garbage collection occurs at times unspecified by your program. Since garbage collection takes time, you might not want it to occur during some time-critical task, or you might want to take advantage of idle time to perform garbage collection and other types of "housekeeping" chores.

There are two methods that are especially important if you have unmanaged code in your project: **AddMemoryPressure( )** and **RemoveMemoryPressure( )**. These are used to indicate that a large amount of unmanaged memory has been allocated or released by the program. They are important because the memory management system has no oversight on unmanaged memory. If a program allocates a large amount of unmanaged memory, then performance might be affected because the system has no way of knowing that free memory has been reduced. By calling **AddMemoryPressure( )** when allocating large amounts of unmanaged memory, you let the CLR know that memory has been reduced. By calling **RemoveMemoryPressure( )**, you let the CLR know the memory has been freed. Remember: **RemoveMemoryPressure( )** must be called only to indicate that memory reported by a call to **AddMemoryPressure( )** has been released.

Method	Meaning
public static void AddMemoryPressure(long *size*)	Indicates that *size* number of bytes of unmanaged memory have been allocated.
public static void Collect( )	Initiates garbage collection.
public static void Collect(int *maxGen*)	Initiates garbage collection for memory with generation numbers of 0 through *maxGen.*
public static void Collect(int *maxGen*, GCCollectionMode *GCMode*)	Initiates garbage collection for memory with generation numbers of 0 through *maxGen* as specified by *GCMode*.
public static int CollectionCount(int *gen*)	Returns the number of garbage collections that have taken place for memory having the generation number specified by *gen.*
public static int GetGeneration(object *o*)	Returns the generation number for the memory referred to by *o.*
public static int GetGeneration(WeakReference *o*)	Returns the generation number for the memory referred to by the weak reference specified by *o.* A weak reference does not prevent the object from being garbage-collected.
public static long GetTotalMemory(bool *collect*)	Returns the total number of bytes currently allocated. If *collect* is true, garbage collection occurs first.
public static void KeepAlive(object *o*)	Creates a reference to *o,* thus preventing it from being garbage collected. This reference ends when **KeepAlive( )** executes.
public static void RemoveMemoryPressure(long *size*)	Indicates that *size* number of bytes of unmanaged memory have been released.
public static void ReRegisterForFinalize(object *o*)	Causes the finalizer (i.e., the destructor) for *o* to be called. This method undoes the effects of **SuppressFinalize( )**.
public static void SuppressFinalize(object *o*)	Prevents the finalizer (i.e., the destructor) for *o* from being called.
public static void WaitForPendingFinalizers( )	Halts execution of the invoking thread until all pending finalizers (i.e., destructors) have been called.

TABLE 21-15 Methods Defined by **GC**

Object

Object is the class that underlies the C# **object** type. The members of **Object** were discussed in Chapter 11, but because of its central role in C#, its methods are repeated in Table 21-16 for your convenience. **Object** defines one constructor, which is shown here:

public Object( )

It constructs an empty object.

The IComparable and IComparable<T> Interfaces

Many classes will need to implement either the **IComparable** or **IComparable<T>** interface because it enables one object to be compared to another by various methods defined by the .NET Framework. Chapter 18 introduced the **IComparable** and **IComparable<T>** interfaces, where they were used to enable two objects of a generic type parameter to be compared. They were also mentioned in the discussion of **Array**, earlier in this chapter. However, because of their importance and applicability to many situations, they are formally examined here.

IComparable is especially easy to implement because it consists of just this one method:

int CompareTo(object *v*)

This method compares the invoking object against the value in *v*. It returns greater than zero if the invoking object is greater than *v*, zero if the two objects are equal, and less than zero if the invoking object is less than *v*.

Method	Purpose
public virtual bool Equals(object *ob*)	Returns true if the invoking object is the same as the one referred to by *object*. Returns false otherwise.
public static bool Equals(object *ob1*, object *ob2*)	Returns true if *ob1* is the same as *ob2*. Returns false otherwise.
protected Finalize( )	Performs shutdown actions prior to garbage collection. In C#, **Finalize( )** is accessed through a destructor.
public virtual int GetHashCode( )	Returns the hash code associated with the invoking object.
public Type GetType( )	Obtains the type of an object at runtime.
protected object MemberwiseClone( )	Makes a "shallow copy" of the object. This is one in which the members are copied, but objects referred to by members are not.
public static bool ReferenceEquals(object *ob1*, object *ob2*)	Returns true if *ob1* and *ob2* refer to the same object. Returns false otherwise.
public virtual string ToString( )	Returns a string that describes the object.

Table 21-16 Methods Defined by **Object**

The generic version of **IComparable** is declared like this:

```
public interface IComparable<T>
```

In this version, the type of data being compared is passed as a type argument to **T**. This causes the declaration of **CompareTo( )** to be changed, as shown next.

```
int CompareTo(T obj)
```

Here, the type of data that **CompareTo( )** operates on can be explicitly specified. This makes **IComparable<T>** type-safe. For this reason, **IComparable<T>** is now preferable to **IComparable**.

The IEquatable<T> Interface

IEquatable<T>is implemented by those classes that need to define how two objects should be compared for equality. It defines only one method, **Equals( )**, which is shown here:

```
bool Equals(T obj)
```

The method returns **true** if *obj* is equal to the invoking object and **false** otherwise.

IEquatable<T> is implemented by several classes and structures in the .NET Framework, including the numeric structures, **Char**, **Int32**, **Boolean**, and **String**.

The IConvertible Interface

The **IConvertible** interface is implemented by all of the value-type structures, **string**, and **DateTime**. It specifies various type conversions. Normally, classes that you create will not need to implement this interface.

The ICloneable Interface

By implementing the **ICloneable** interface, you enable a copy of an object to be made. **ICloneable** defines only one method, **Clone( )**, which is shown here:

```
object Clone( )
```

This method makes a copy of the invoking object. How you implement **Clone( )** determines how the copy is made. In general, there are two types of copies: deep and shallow. When a deep copy is made, the copy and original are completely independent. Thus, if the original object contained a reference to another object *O*, then a copy of *O* will also be made. In a shallow copy, members are copied, but objects referred to by members are not. If an object refers to some other object *O*, then after a shallow copy, both the copy and the original will refer to the same *O*, and any changes to *O* affect both the copy and the original. Usually, you will implement **Clone( )** so that it performs a deep copy. Shallow copies can be made by using **MemberwiseClone( )**, which is defined by **Object**.

Here is an example that illustrates **ICloneable**. It creates a class called **Test** that contains a reference to an object of a class called **X**. **Test** uses **Clone( )** to create a deep copy.

```
// Demonstrate ICloneable.

using System;

class X {
```

```
  public int a;

  public X(int x) { a = x; }
}

class Test : ICloneable {
  public X o;
  public int b;

  public Test(int x, int y) {
    o = new X(x);
    b = y;
  }

  public void Show(string name) {
    Console.Write(name + " values are ");
    Console.WriteLine("o.a: {0}, b: {1}", o.a, b);
  }

  // Make a deep copy of the invoking object.
  public object Clone() {
    Test temp = new Test(o.a, b);
    return temp;
  }

}

class CloneDemo {
  static void Main() {
    Test ob1 = new Test(10, 20);

    ob1.Show("ob1");

    Console.WriteLine("Make ob2 a clone of ob1.");
    Test ob2 = (Test) ob1.Clone();

    ob2.Show("ob2");

    Console.WriteLine("Changing ob1.o.a to 99 and ob1.b to 88.");
    ob1.o.a = 99;
    ob1.b = 88;

    ob1.Show("ob1");
    ob2.Show("ob2");
  }
}
```

The output is shown here:

```
ob1 values are o.a: 10, b: 20
Make ob2 a clone of ob1.
ob2 values are o.a: 10, b: 20
Changing ob1.o.a to 99 and ob1.b to 88.
ob1 values are o.a: 99, b: 88
ob2 values are o.a: 10, b: 20
```

As the output shows, **ob2** is a clone of **ob1**, but **ob1** and **ob2** are completely separate objects. Changing one does not affect the other. This is accomplished by constructing a new **Test** object, which allocates a new **X** object for the copy. The new **X** instance is given the same value as the **X** object in the original.

To implement a shallow copy, simply have **Clone( )** call **MemberwiseClone( )** defined by **Object**. For example, try changing **Clone( )** in the preceding program as shown here:

```
// Make a shallow copy of the invoking object.
public object Clone() {
  Test temp = (Test) MemberwiseClone();
  return temp;
}
```

After making this change, the output of the program will look like this:

```
ob1 values are o.a: 10, b: 20
Make ob2 a clone of ob1.
ob2 values are o.a: 10, b: 20
Changing ob1.o.a to 99 and ob1.b to 88.
ob1 values are o.a: 99, b: 88
ob2 values are o.a: 99, b: 20
```

Notice that **o** in **ob1** and **o** in **ob2** both refer to the same **X** object. Changing one affects both. Of course, the **int** field **b** in each is still separate because the value types are not accessed via references.

IFormatProvider and IFormattable

The **IFormatProvider** interface defines one method called **GetFormat( )**, which returns an object that controls the formatting of data into a human-readable string. The general form of **GetFormat( )** is shown here:

object GetFormat(Type *fmt*)

Here, *fmt* specifies the format object to obtain.

The **IFormattable** interface supports the formatting of human-readable output. **IFormattable** defines this method:

string ToString(string *fmt*, IFormatProvider *fmtpvdr*)

Here, *fmt* specifies formatting instructions and *fmtpvdr* specifies the format provider.

NOTE *Formatting is described in detail in Chapter 22.*

22
CHAPTER

Strings and Formatting

This chapter examines the **String** class, which underlies C#'s **string** type. As all programmers know, string handling is a part of almost any program. For this reason, the **String** class defines an extensive set of methods, properties, and fields that give you detailed control over the construction and manipulation of strings. Closely related to string handling is the formatting of data into its human-readable form. Using the formatting subsystem, you can format the C# numeric types, date and time, and enumerations.

Strings in C#

An overview of C#'s string handling was presented in Chapter 7, and that discussion is not repeated here. However, it is worthwhile to review how strings are implemented in C# before examining the **String** class.

In all computer languages, a *string* is a sequence of characters, but precisely how such a sequence is implemented varies from language to language. In some computer languages, such as C++, strings are arrays of characters, but this is not the case with C#. Instead, C# strings are objects of the built-in **string** data type. Thus, **string** is a reference type. Moreover, **string** is C#'s name for **System.String**, the standard .NET string type. Thus, a C# string has access to all of the methods, properties, fields, and operators defined by **String**.

Once a string has been created, the character sequence that comprises a string cannot be altered. This restriction allows C# to implement strings more efficiently. Though this restriction probably sounds like a serious drawback, it isn't. When you need a string that is a variation on one that already exists, simply create a new string that contains the desired changes, and discard the original string if it is no longer needed. Because unused string objects are automatically garbage-collected, you don't need to worry about what happens to the discarded strings. It must be made clear, however, that **string** reference variables may, of course, change the object to which they refer. It is just that the character sequence of a specific **string** object cannot be changed after it is created.

To create a string that can be changed, C# offers a class called **StringBuilder**, which is in the **System.Text** namespace. For most purposes, however, you will want to use **string**, not **StringBuilder**.

The String Class

String is defined in the **System** namespace. It implements the **IComparable**, **IComparable<string>**, **ICloneable**, **IConvertible**, **IEnumerable**, **IEnumerable<char>**, and **IEquatable<string>** interfaces. **String** is a sealed class, which means that it cannot be inherited. **String** provides string-handling functionality for C#. It underlies C#'s built-in **string** type and is part of the .NET Framework. The next few sections examine **String** in detail.

The String Constructors

The **String** class defines several constructors that allow you to construct a string in a variety of ways. To create a string from a character array, use one of these constructors:

public String(char[ ] *chrs*)
public String(char[ ] *chrs*, int *start*, int *count*)

The first form constructs a string that contains the characters in *chrs.* The second form uses *count* characters from *chrs,* beginning at the index specified by *start.*

You can create a string that contains a specific character repeated a number of times using this constructor:

public String(char *ch*, int *count*)

Here, *ch* specifies the character that will be repeated *count* times.

You can construct a string given a pointer to a character array using one of these constructors:

public String(char* *chrs*)
public String(char* *chrs*, int *start*, int *count*)

The first form constructs a string that contains the characters pointed to by *chrs.* It is assumed that *chrs* points to a null-terminated array, which is used in its entirety. The second form uses *count* characters from the array pointed to by *chrs,* beginning at the index specified by *start.* Because they use pointers, these constructors can be used only in unsafe code.

You can construct a string given a pointer to an array of bytes using one of these constructors:

public String(sbyte* *chrs*)
public String(sbyte* *chrs*, int *start*, int *count*)
public String(sbyte* *chrs*, int *start*, int *count*, Encoding *en*)

The first form constructs a string that contains the bytes pointed to by *chrs.* It is assumed that *chrs* points to a null-terminated array, which is used in its entirety. The second form uses *count* characters from the array pointed to by *chrs,* beginning at the index specified by *start.* The third form lets you specify how the bytes are encoded. The default encoding is **ASCIIEncoding**. The **Encoding** class is in the **System.Text** namespace. Because they use pointers, these constructors can be used only in unsafe code.

A string literal automatically creates a string object. For this reason, a **string** object is often initialized by assigning it a string literal, as shown here:

```
string str = "a new string";
```

The String Field, Indexer, and Property

The **String** class defines one field, shown here:

public static readonly string Empty

Empty specifies an empty string, which is a string that contains no characters. This differs from a null **String** reference, which simply refers to no object.

There is one read-only indexer defined for **String**, which is shown here:

public char this[int idx] { get; }

This indexer allows you to obtain the character at a specified index. Like arrays, the indexing for strings begins at zero. Since **String** objects are immutable, it makes sense that **String** supports a read-only indexer.

There is one read-only property:

public int Length { get; }

Length returns the number of characters in the string.

The String Operators

The **String** class overloads two operators: **= =** and **!=**. To test two strings for equality, use the **= =** operator. Normally, when the **= =** operator is applied to object references, it determines if both references refer to the same object. This differs for objects of type **String**. When the **= =** is applied to two **String** references, the contents of the strings, themselves, are compared for equality. The same is true for the **!=** operator: When comparing **String** objects, the contents of the strings are compared. However, the other relational operators, such as **<** or **>=**, compare the references, just like they do for other types of objects. To determine if one string is greater than or less than another, use the **Compare( )** method defined by **String**.

The String Methods

The **String** class defines a large number of methods, and many of the methods have two or more overloaded forms. For this reason, it is neither practical nor useful to list them all. Instead, several of the more commonly used methods will be presented, along with examples that illustrate them.

Comparing Strings

Perhaps the most frequently used string-handling operation is the comparison of one string to another. Because of its importance, **String** provides a wide array of comparison methods. These are shown in Table 22-1. Be aware that string comparisons are sensitive to cultural differences. Comparison methods that do not pass cultural information use the currently selected cultural settings.

Method	Description
public static int Compare(string *str1*, string *str2*)	Compares the string referred to by *str1* with *str2.* Returns greater than zero if *str1* is greater than *str2,* less than zero if *str1* is less than *str2,* and zero if *str1* and *str2* are equal.
public static int Compare(string *str1*, string *str2*, bool *ignoreCase*)	Compares the string referred to by *str1* with *str2.* Returns greater than zero if *str1* is greater than *str2,* less than zero if *str1* is less than *str2,* and zero if *str1* and *str2* are equal. If *ignoreCase* is **true**, the comparison ignores case differences. Otherwise, case differences matter.
public static int Compare(string *str1*, string *str2*, StringComparison *how*)	Compares the string referred to by *str1* with *str2.* Returns greater than zero if *str1* is greater than *str2,* less than zero if *str1* is less than *str2,* and zero if *str1* and *str2* are equal. How the comparison is performed is specified by *how.*
public static int Compare(string *str1*, string *str2*, bool *ignoreCase*, CultureInfo *ci*)	Compares the string referred to by *str1* with *str2* using the cultural information passed in *ci.* Returns greater than zero if *str1* is greater than *str2,* less than zero if *str1* is less than *str2,* and zero if *str1* and *str2* are equal. If *ignoreCase* is **true**, the comparison ignores case differences. Otherwise, case differences matter. The **CultureInfo** class is defined in the **System.Globalization** namespace.
public static int Compare(string *str1*, int *start1*, string *str2*, int *start2*, int *count*)	Compares portions of the strings referred to by *str1* and *str2.* The comparison begins at *str1*[*start1*] and *str2*[*start2*] and runs for *count* characters. Returns greater than zero if *str1* is greater than *str2,* less than zero if *str1* is less than *str2,* and zero if *str1* and *str2* are equal.
public static int Compare(string *str1*, int *start1*, string *str2*, int *start2*, int *count*, bool *ignoreCase*)	Compares portions of the strings referred to by *str1* and *str2.* The comparison begins at *str1*[*start1*] and *str2*[*start2*] and runs for *count* characters. Returns greater than zero if *str1* is greater than *str2,* less than zero if *str1* is less than *str2,* and zero if *str1* and *str2* are equal. If *ignoreCase* is **true**, the comparison ignores case differences. Otherwise, case differences matter.
public static int Compare(string *str1*, int *start1*, string *str2*, int *start2*, int *count*, StringComparison *how*)	Compares portions of the strings referred to by *str1* and *str2.* The comparison begins at *str1*[*start1*] and *str2*[*start2*] and runs for *count* characters. Returns greater than zero if *str1* is greater than *str2,* less than zero if *str1* is less than *str2,* and zero if *str1* and *str2* are equal. How the comparison is performed is specified by *how.*

TABLE 22-1 The **String** Comparison Methods

Method	Description
public static int Compare(string *str1*, int *start1*, string *str2*, int *start2*, int *count*, bool *ignoreCase*, CultureInfo *ci*)	Compares portions of the strings referred to by *str1* and *str2* using the cultural information passed in *ci*. The comparison begins at *str1*[*start1*] and *str2*[*start2*] and runs for *count* characters. Returns greater than zero if *str1* is greater than *str2*, less than zero if *str1* is less than *str2*, and zero if *str1* and *str2* are equal. If *ignoreCase* is true, the comparison ignores case differences. Otherwise, case differences matter. The **CultureInfo** class is defined in the **System.Globalization** namespace.
public static int CompareOrdinal(string *str1*, string *str2*)	Compares the string referred to by *str1* with *str2* independently of culture, region, or language. Returns greater than zero if *str1* is greater than *str2*, less than zero if *str1* is less than *str2*, and zero if *str1* and *str2* are equal.
public static int CompareOrdinal(string *str1*, int *start1*, string *str2*, int *start2*, int *count*)	Compares portions of the strings referred to by *str1* and *str2* independently of culture, region, or language. The comparison begins at *str1*[*start1*] and *str2*[*start2*] and runs for *count* characters. Returns greater than zero if *str1* is greater than *str2*, less than zero if *str1* is less than *str2*, and zero if *str1* and *str2* are equal.
public int CompareTo(object *str*)	Compares the invoking string with *str*. Returns greater than zero if the invoking string is greater than *str*, less than zero if the invoking string is less than *str*, and zero if the two are equal.
public int CompareTo(string *str*)	Compares the invoking string with *str*. Returns greater than zero if the invoking string is greater than *str*, less than zero if the invoking string is less than *str*, and zero if the two are equal.

TABLE 22-1 The **String** Comparison Methods *(continued)*

Of the comparison methods, the **Compare( )** method is the most versatile. It can compare two strings in their entirety or in parts. It can use case-sensitive comparisons or ignore case. In general, string comparisons use dictionary order to determine whether one string is greater than, equal to, or less than another. You can also specify cultural information that governs the comparison. The following program demonstrates several versions of **Compare( )**:

```
// Compare strings.
using System;

class CompareDemo {
  static void Main() {
    string str1 = "one";
    string str2 = "one";
```

```
    string str3 = "ONE";
    string str4 = "two";
    string str5 = "one, too";


    if(String.Compare(str1, str2) == 0)
      Console.WriteLine(str1 + " and " + str2 +
                        " are equal.");
    else
      Console.WriteLine(str1 + " and " + str2 +
                        " are not equal.");

    if(String.Compare(str1, str3) == 0)
      Console.WriteLine(str1 + " and " + str3 +
                        " are equal.");
    else
      Console.WriteLine(str1 + " and " + str3 +
                        " are not equal.");

    if(String.Compare(str1, str3, true) == 0)
      Console.WriteLine(str1 + " and " + str3 +
                        " are equal ignoring case.");
    else
      Console.WriteLine(str1 + " and " + str3 +
                        " are not equal ignoring case.");

     if(String.Compare(str1, str5) == 0)
      Console.WriteLine(str1 + " and " + str5 +
                        " are equal.");
    else
      Console.WriteLine(str1 + " and " + str5 +
                        " are not equal.");

    if(String.Compare(str1, 0, str5, 0, 3) == 0)
      Console.WriteLine("First part of " + str1 + " and " +
                        str5 + " are equal.");
    else
      Console.WriteLine("First part of " + str1 + " and " +
                        str5 + " are not equal.");

    int result = String.Compare(str1, str4);
    if(result < 0)
      Console.WriteLine(str1 + " is less than " + str4);
    else if(result > 0)
      Console.WriteLine(str1 + " is greater than " + str4);
    else
      Console.WriteLine(str1 + " equals " + str4);
  }
}
```

The output is shown here:

```
one and one are equal.
one and ONE are not equal.
one and ONE are equal ignoring case.
one and one, too are not equal.
First part of one and one, too are equal.
one is less than two
```

Using The StringComparison Enumeration

In Table 22-1, notice the two **Compare( )** methods that take a parameter of type **StringComparison**. These versions are shown here:

public static int Compare(string *str1*, string *str2*, StringComparison *how*)

public static int Compare(string *str1*, int *start1*, string *str2*, int *start2*,
int *count*, StringComparison *how*)

For each version, the *how* parameter specifies how the comparison of *str1* with *str2* takes place. **StringComparison** is an enumeration that defines the values shown in Table 22-2. Using these values, it is possible to craft a comparison that meets the specific needs of your application. Thus, the addition of the **StringComparison** parameter expands the capabilities of **Compare( )**.

One particularly good use of the **StringComparison** form of **Compare( )** is to compare a string against an invariant file name or password. For example, imagine a situation in which the user must enter the password **we~23&blx$**. This password is the same no matter what cultural settings are in effect. Thus, you want to compare the string entered by the user to the password without cultural differences affecting the comparison. One way to do this is

Value	Description
CurrentCulture	Comparisons are performed using the currently active cultural settings.
CurrentCultureIgnoreCase	Case-insensitive comparisons are performed using the currently active cultural settings.
InvariantCulture	Comparisons are performed using an invariant (that is, universal and unchanging) culture.
InvariantCultureIngoreCase	Case-insensitive comparisons are performed using an invariant (that is, universal and unchanging) culture.
Ordinal	Comparisons are performed using the ordinal values of the characters in the string. Thus, dictionary-order may not result and cultural conventions are ignored.
OrdinalIgnoreCase	Case-insensitive comparisons are performed using the ordinal values of the characters in the string. Thus, dictionary-order may not result and cultural conventions are ignored.

Table 22-2 The **StringComparison** Enumeration Values

to specify **StringComparison.InvariantCulture** for the *how* parameter in which case all cultural differences are avoided. This approach is demonstrated by the following program:

```
// Compare strings using StringComparison enumeration.

using System;

class StrCompDemo {
  static void Main() {
    // Note: Never embed a password in real code.
    // This is for demonstration purposes only.
    string pswd = "we~23&blx$";

    string str;

    Console.WriteLine("Enter password: ");
    str = Console.ReadLine();

    // Compare using invariant culture.
    if(String.Compare(pswd, str,
                      StringComparison.InvariantCulture) == 0)
      Console.WriteLine("Password accepted.");
    else
      Console.WriteLine("Password invalid.");
  }
}
```

Concatenating Strings

There are two ways to concatenate (join together) two or more strings. First, you can use the **+** operator, as demonstrated in Chapter 7. Second, you can use one of the various concatenation methods defined by **String**. Although using **+** is the easiest approach in many cases, the concatenation methods give you an alternative.

The method that performs concatenation is called **Concat( )**. One of its most commonly used forms is shown here:

public static string Concat(string *str1*, string *str2*)

This method returns a string that contains *str2* concatenated to the end of *str1*. Another form of **Concat( )**, shown here, concatenates three strings:

public static string Concat(string *str1*, string *str2*, string *str3*)

In this version, a string that contains the concatenation of *str1*, *str2*, and *str3* is returned. There is also a form that concatenates four strings:

public static string Concat(string *str1*, string *str2*, string *str3*, string *str4*)

This version returns the concatenation of all four strings.

The version of **Concat( )** shown next concatenates an arbitrary number of strings:

public static string Concat(params string[ ] *strs*)

Here, *strs* refers to a variable number of arguments that are concatenated, and the result is returned. Because this version of **Concat( )** can be used to concatenate any number of strings, including two, three, or four strings, you might wonder why the other forms just shown

exist. The reason is efficiency; passing up to four arguments is more efficient than using a variable-length argument list.

The following program demonstrates the variable-length argument version of **Concat( )**:

```
// Demonstrate Concat().

using System;

class ConcatDemo {
  static void Main() {

    string result = String.Concat("This ", "is ", "a ",
                                  "test ", "of ", "the ",
                                  "String ", "class.");

    Console.WriteLine("result: " + result);

  }
}
```

The output is shown here:

```
result: This is a test of the String class.
```

There are also versions of the **Concat( )** method that take **object** references, rather than **string** references. These obtain the string representation of the objects with which they are called and return a string containing the concatenation of those strings. (The string representations are obtained by calling **ToString( )** on the objects.) These versions of **Concat( )** are shown here:

public static string Concat(object *v1*)
public static string Concat(object *v1*, object *v2*)
public static string Concat(object *v1*, object *v2*, object *v3*)
public static string Concat(object *v1*, object *v2*, object *v3*, object *v4*)
public static string Concat(params object[ ] *v*)

The first method simply returns the string equivalent of *v1*. The other methods return a string that contains the concatenation of their arguments. The **object** forms of **Concat( )** are very convenient because they let you avoid having to manually obtain string representations prior to concatenation. To see how useful these methods can be, consider the following program:

```
// Demonstrate the object form of Concat().

using System;

class MyClass {
  public static int Count = 0;

  public MyClass() { Count++; }
}

class ConcatDemo {
  static void Main() {
```

```
    string result = String.Concat("The value is " + 19);
    Console.WriteLine("result: " + result);

    result = String.Concat("hello ", 88, " ", 20.0, " ",
                           false, " ",  23.45M);
    Console.WriteLine("result: " + result);

    MyClass mc = new MyClass();

    result = String.Concat(mc, " current count is ",
                           MyClass.Count);
    Console.WriteLine("result: " + result);
  }
}
```

The output is shown here:

```
result: The value is 19
result: hello 88 20 False 23.45
result: MyClass current count is 1
```

In this example, **Concat( )** concatenates the string representations of various types of data. For each argument, the **ToString( )** method associated with that argument is called to obtain a string representation. Thus, in this call to **Concat( )**

```
string result = String.Concat("The value is " + 19);
```

Int32.ToString( ) is invoked to obtain the string representation of the integer value 19. **Concat( )** then concatenates the strings and returns the result.

Also notice how an object of the user-defined class **MyClass** can be used in this call to **Concat( )**:

```
result = String.Concat(mc, " current count is ",
                       MyClass.Count);
```

In this case, the string representation of **mc**, which is of type **MyClass**, is returned. By default, this is simply its class name. However, if you override the **ToString( )** method, then **MyClass** can return a different string. For example, try adding this version of **ToString( )** to **MyClass** in the preceding program:

```
public override string ToString() {
  return "An object of type MyClass";
}
```

When this version is used, the last line in the output will be

```
result: An object of type MyClass current count is 1
```

Searching a String

String offers many methods that allow you to search a string. For example, you can search for either a substring or a character. You can also search for the first or last occurrence of either.

To search for the first occurrence of a character or substring, use the **IndexOf( )** method. Here are two of its forms:

public int IndexOf(char *ch*)
public int IndexOf(String *str*)

The first form returns the index of the first occurrence of the character *ch* within the invoking string. The second form returns the first occurrence of the string *str.* Both return –1 if the item is not found.

To search for the last occurrence of a character or substring, use the **LastIndexOf( )** method. Here are two of its forms:

public int LastIndexOf(char *ch*)
public int LastIndexOf(string *str*)

The first form returns the index of the last occurrence of the character *ch* within the invoking string. The second form returns the index of the last occurrence of the string *str.* Both return –1 if the item is not found.

String offers two interesting supplemental search methods: **IndexOfAny( )** and **LastIndexOfAny( )**. These search for the first or last character that matches any of a set of characters. Here are their simplest forms:

public int IndexOfAny(char[ ] *a*)
public int LastIndexOfAny(char[ ] *a*)

IndexOfAny( ) returns the index of the first occurrence of any character in *a* that is found within the invoking string. **LastIndexOfAny( )** returns the index of the last occurrence of any character in *a* that is found within the invoking string. Both return –1 if no match is found.

When working with strings, it is often useful to know if a string begins with or ends with a given substring. To accomplish these tasks, use the **StartsWith( )** and **EndsWith( )** methods. Here are their two simplest forms:

public bool StartsWith(string *str*)
public bool EndsWith(string *str*)

StartsWith( ) returns true if the invoking string begins with the string passed in *str.* **EndsWith( )** returns true if the invoking string ends with the string passed in *str.* Both return false on failure.

Here is a program that demonstrates several of the string search methods:

```
// Search strings.

using System;

class StringSearchDemo {
  static void Main() {
    string str = "C# has powerful string handling.";
    int idx;

    Console.WriteLine("str: " + str);

    idx = str.IndexOf('h');
    Console.WriteLine("Index of first 'h': " + idx);
```

```
    idx = str.LastIndexOf('h');
    Console.WriteLine("Index of last 'h': " + idx);

    idx = str.IndexOf("ing");
    Console.WriteLine("Index of first \"ing\": " + idx);

    idx = str.LastIndexOf("ing");
    Console.WriteLine("Index of last \"ing\": " + idx);

    char[] chrs = { 'a', 'b', 'c' };
    idx = str.IndexOfAny(chrs);
    Console.WriteLine("Index of first 'a', 'b', or 'c': " + idx);

    if(str.StartsWith("C# has"))
      Console.WriteLine("str begins with \"C# has\"");

    if(str.EndsWith("ling."))
      Console.WriteLine("str ends with \"ling.\"");
  }
}
```

The output from the program is shown here:

```
str: C# has powerful string handling.
Index of first 'h': 3
Index of last 'h': 23
Index of first "ing": 19
Index of last "ing": 28
Index of first 'a', 'b', or 'c': 4
str begins with "C# has"
str ends with "ling."
```

A string search method that you will find useful in many circumstances is **Contains( )**. Its general form is shown here:

public bool Contains(string *str*)

It returns true if the invoking string contains the string specified by *str* and false otherwise. This method is especially useful when all you need to know is if a specific substring exists within another string. Here is an example that demonstrates its use:

```
// Demonstrate Contains().

using System;

class ContainsDemo {
  static void Main() {
    string str = "C# combines power with performance.";

    if(str.Contains("power"))
      Console.WriteLine("The sequence power was found.");

    if(str.Contains("pow"))
      Console.WriteLine("The sequence pow was found.");
```

```
    if(!str.Contains("powerful"))
      Console.WriteLine("The sequence powerful was not found.");
  }
}
```

The output is shown here:

```
The sequence power was found.
The sequence pow was found.
The sequence powerful was not found.
```

As the output shows, **Contains( )** searches for a matching sequence, not for whole words. Thus, both "pow" and "power" are found. However, since there is no sequence that matches "powerful", it is (correctly) not found.

Several of the search methods have additional forms that allow you to begin a search at a specified index or to specify a range to search within. All versions of the **String** search methods are shown in Table 22-3.

Method	Description
public bool Contains(string *str*)	Returns true if the invoking string contains the string specified by *str.* False is returned if *str* is not found.
public bool EndsWith(string *str*)	Returns true if the invoking string ends with the string passed in *str.* Otherwise, false is returned.
public bool EndsWith(string *str*, StringComparison *how*)	Returns true if the invoking string ends with the string passed in *str.* Otherwise, false is returned. How the search is performed is specified by *how*.
public bool EndsWith(string *str*, bool *ignoreCase*, CultureInfo *ci*)	Returns true if the invoking string ends with the string passed in *str.* Otherwise, false is returned. If *ignoreCase* is **true**, the search ignores case differences. Otherwise, case differences matter. The search is conducted using the cultural information passed in *ci.*
public int IndexOf(char *ch*)	Returns the index of the first occurrence of *ch* within the invoking string. Returns –1 if *ch* is not found.
public int IndexOf(string *str*)	Returns the index of the first occurrence of *str* within the invoking string. Returns –1 if *str* is not found.
public int IndexOf(char *ch*, int *start*)	Returns the index of the first occurrence of *ch* within the invoking string. Searching begins at the index specified by *start.* Returns –1 if *ch* is not found.
public int IndexOf(string *str*, int *start*)	Returns the index of the first occurrence of *str* within the invoking string. Searching begins at the index specified by *start.* Returns –1 if *str* is not found.

Table 22-3 The Search Methods Offered by **String**

Method	Description
public int IndexOf(char *ch*, int *start*, int *count*)	Returns the index of the first occurrence of *ch* within the invoking string. Searching begins at the index specified by *start* and runs for *count* elements. Returns –1 if *ch* is not found.
public int IndexOf(string *str*, int *start*, int *count*)	Returns the index of the first occurrence of *str* within the invoking string. Searching begins at the index specified by *start* and runs for *count* elements. Returns –1 if *str* is not found.
public int IndexOf(string *str*, StringComparison *how*)	Returns the index of the first occurrence of *str* within the invoking string. How the search is performed is specified by *how.* Returns –1 if *str* is not found.
public int IndexOf(string *str*, int *start*, StringComparison *how*)	Returns the index of the first occurrence of *str* within the invoking string. Searching begins at the index specified by *start.* How the search is performed is specified by *how*. Returns –1 if *str* is not found.
public int IndexOf(string *str*, int *start*, int *count*, StringComparison *how*)	Returns the index of the first occurrence of *str* within the invoking string. Searching begins at the index specified by *start* and runs for *count* elements. How the search is performed is specified by *how*. Returns –1 if *ch* is not found.
public int LastIndexOf(char *ch*)	Returns the index of the last occurrence of *ch* within the invoking string. Returns –1 if *ch* is not found.
public int LastIndexOf(string *str*)	Returns the index of the last occurrence of *str* within the invoking string. Returns –1 if *str* is not found.
public int LastIndexOf(char *ch*, int *start*)	Returns the index of the last occurrence of *ch* within a range of the invoking string. The search proceeds in reverse order, beginning at the index specified by *start* and stopping at 0. Returns –1 if the *ch* is not found.
public int LastIndexOf(string *str*, int *start*)	Returns the index of the last occurrence of *str* within a range of the invoking string. The search proceeds in reverse order, beginning at the index specified by *start* and stopping at 0. Returns –1 if *str* is not found.
public int LastIndexOf(char *ch*, int *start*, int *count*)	Returns the index of the last occurrence of *ch* within the invoking string. The search proceeds in reverse order, beginning at the index specified by *start* and running for *count* elements. Returns –1 if *ch* is not found.

Table 22-3 The Search Methods Offered by **String** *(continued)*

Method	Description
public int LastIndexOf(string *str*, int *start*, int *count*)	Returns the index of the last occurrence of *str* within the invoking string. The search proceeds in reverse order, beginning at the index specified by *start* and running for *count* elements. Returns –1 if *str* is not found.
public int LastIndexOf(string *str*, StringComparison *how*)	Returns the index of the last occurrence of *str* within the invoking string. How the search is performed is specified by *how*. Returns –1 if *str* is not found.
public int LastIndexOf(string *str*, int *start*, StringComparison *how*)	Returns the index of the last occurrence of *str* within a range of the invoking string. The search proceeds in reverse order, beginning at the index specified by *start* and stopping at 0. How the search is performed is specified by *how.* Returns –1 if *str* is not found.
public int LastIndexOf(string *str*, int *start*, int *count*, StringComparison *how*)	Returns the index of the last occurrence of *str* within the invoking string. The search proceeds in reverse order, beginning at the index specified by *start* and running for *count* elements. How the search is performed is specified by *how.* Returns –1 if *str* is not found.
public bool StartsWith(string *str*)	Returns true if the invoking string begins with the string passed in *str.* Otherwise, false is returned.
public bool StartsWith(string *str*, StringComparison *how*)	Returns true if the invoking string begins with the string passed in *str.* Otherwise, false is returned. How the search is performed is specified by *how*.
public bool StartsWith(string *str*, bool *ignoreCase*, CultureInfo *ci*)	Returns true if the invoking string begins with the string passed in *str.* Otherwise, false is returned. If *ignoreCase* is **true**, the search ignores case differences. Otherwise, case differences matter. The search is conducted using the cultural information passed in *ci.*

Table 22-3 The Search Methods Offered by **String** *(continued)*

Splitting and Joining Strings

Two fundamental string-handling operations are split and join. A *split* decomposes a string into its constituent parts. A *join* constructs a string from a set of parts. To split a string, **String** defines **Split( )**. To join a set of strings, **String** provides **Join( )**.

There are several versions of **Split( )**. Two commonly used forms, which have been available since C# 1.0, are shown here:

public string[ ] Split(params char[ ] *seps*)
public string[ ] Split(params char[ ] *seps*, int *count*)

The first form splits the invoking string into pieces and returns an array containing the substrings. The characters that delimit each substring are passed in *seps*. If *seps* is null or refers to an empty string, then whitespace is used as the separator. In the second form, no more than *count* substrings will be returned.

The two forms of the **Join( )** method are shown here:

public static string Join(string *sep*, string[ ] *strs*)
public static string Join(string *sep*, string[ ] *strs*, int *start*, int *count*)

The first form returns a string that contains the concatenation of the strings in *strs*. The second form returns a string that contains the concatenation of *count* strings in *strs*, beginning at *strs*[*start*]. For both versions, each string is separated from the next by the string specified by *sep*.

The following program demonstrates **Split( )** and **Join( )**:

```
// Split and join strings.

using System;

class SplitAndJoinDemo {
  static void Main() {
    string str = "One if by land, two if by sea.";
    char[] seps = {' ', '.', ',' };

    // Split the string into parts.
    string[] parts = str.Split(seps);
    Console.WriteLine("Pieces from split: ");
    for(int i=0; i < parts.Length; i++)
      Console.WriteLine(parts[i]);

    // Now, join the parts.
    string whole = String.Join(" | ", parts);
    Console.WriteLine("Result of join: ");
    Console.WriteLine(whole);
  }
}
```

Here is the output:

```
Pieces from split:
One
if
by
land

two
if
by
sea

Result of join:
One | if | by | land |  | two | if | by | sea |
```

There is one important thing to notice in this output: the empty string that occurs between "land" and "two". This is caused by the fact that in the original string, the word "land" is followed by a comma and a space, as in "land, two". However, both the comma and the space are specified as separators. Thus, when this string is split, the empty string that exists between the two separators (the comma and the space) is returned.

There are several additional forms of **Split( )** that take a parameter of type **StringSplitOptions**. This parameter controls whether empty strings are part of the resulting split. Here are these forms of **Split( )**:

public string[ ] Split(params char[ ] *seps*, StringSplitOptions *how*)
public string[ ] Split(string[ ] *seps*, StringSplitOptions *how*)
public string[ ] Split(params char[ ] *seps*, int *count*, StringSplitOptions *how*)
public string[ ] Split(string[ ] *seps*, int *count*, StringSplitOptions *how*)

The first two forms split the invoking string into pieces and return an array containing the substrings. The characters that delimit each substring are passed in *seps*. If *seps* is null, then whitespace is used as the separator. In the third and fourth forms, no more than *count* substrings will be returned. For all versions, the value of *how* determines how to handle empty strings that result when two separators are adjacent to each other. The **StringSplitOptions** enumeration defines only two values: **None** and **RemoveEmptyEntries**. If *how* is **None**, then empty strings are included in the result (as the previous program showed). If *how* is **RemoveEmptyEntries**, empty strings are excluded from the result.

To understand the effects of removing empty entries, try replacing this line in the preceding program:

```
string[] parts = str.Split(seps);
```

with the following:

```
string[] parts = str.Split(seps, StringSplitOptions.RemoveEmptyEntries);
```

When you run the program, the output will be as shown next:

```
Pieces from split:
One
if
by
land
two
if
by
sea
Result of join:
One | if | by | land | two | if | by | sea
```

As you can see, the empty string that previously resulted because of the combination of the comma and space after "land" has been removed.

Splitting a string is an important string-manipulation procedure because it is often used to obtain the individual *tokens* that comprise the string. For example, a database program might use **Split( )** to decompose a query such as "show me all balances greater than 100"

into its individual parts, such as "show" and "100". In the process, the separators are removed. Thus, "show" (without any leading or trailing spaces) is obtained, not " show". The following program illustrates this concept. It tokenizes strings containing binary mathematical operations, such as 10 + 5. It then performs the operation and displays the result.

```
// Tokenize strings.

using System;

class TokenizeDemo {
  static void Main() {
    string[] input = {
                      "100 + 19",
                      "100 / 3.3",
                      "-3 * 9",
                      "100 - 87"
                     };
    char[] seps = {' '};

    for(int i=0; i < input.Length; i++) {
      // split string into parts
      string[] parts = input[i].Split(seps);
      Console.Write("Command: ");
      for(int j=0; j < parts.Length; j++)
        Console.Write(parts[j] + " ");

      Console.Write(", Result: ");
      double n = Double.Parse(parts[0]);
      double n2 = Double.Parse(parts[2]);

      switch(parts[1]) {
        case "+":
          Console.WriteLine(n + n2);
          break;
        case "-":
          Console.WriteLine(n - n2);
          break;
        case "*":
          Console.WriteLine(n * n2);
          break;
        case "/":
          Console.WriteLine(n / n2);
          break;
      }
    }
  }
}
```

Here is the output:

```
Command: 100 + 19 , Result: 119
Command: 100 / 3.3 , Result: 30.3030303030303
Command: -3 * 9 , Result: -27
Command: 100 - 87 , Result: 13
```

Padding and Trimming Strings

Sometimes you will want to remove leading and trailing spaces from a string. This type of operation, called *trimming*, is often needed by command processors. For example, a database might recognize the word "print." However, a user might enter this command with one or more leading or trailing spaces. Any such spaces must be removed before the string can be recognized by the database. Conversely, sometimes you will want to pad a string with spaces so that it meets some minimal length. For example, if you are preparing formatted output, you might need to ensure that each line is of a certain length in order to maintain alignment. Fortunately, C# includes methods that make these types of operations easy.

To trim a string, use one of these **Trim( )** methods:

public string Trim( )
public string Trim(params char[ ] *chrs*)

The first form removes leading and trailing whitespace from the invoking string. The second form removes leading and trailing occurrences of the characters specified by *chrs.* In both cases, the resulting string is returned.

You can pad a string by adding characters to either the left or the right side of the string. To pad a string on the left, use one of the methods shown here:

public string PadLeft(int *len*)
public string PadLeft(int *len*, char *ch*)

The first form adds spaces on the left as needed to the invoking string so that its total length equals *len.* The second form adds the character specified by *ch* as needed to the invoking string so that its total length equals *len.* In both cases, the resulting string is returned. If *len* is less than the length of the invoking string, a copy of the invoking string is returned unaltered.

To pad a string to the right, use one of these methods:

public string PadRight(int *len*)
public string PadRight(int *len*, char *ch*)

The first form adds spaces on the right as needed to the invoking string so that its total length equals *len.* The second form adds the characters specified by *ch* as needed to the invoking string so that its total length equals *len.* In both cases, the resulting string is returned. If *len* is less than the length of the invoking string, a copy of the invoking string is returned unaltered.

The following program demonstrates trimming and padding:

```
// Trimming and padding.

using System;

class TrimPadDemo {
  static void Main() {
    string str = "test";

    Console.WriteLine("Original string: " + str);

    // Pad on left with spaces.
    str = str.PadLeft(10);
    Console.WriteLine("|" + str + "|");
```

```
    // Pad on right with spaces.
    str = str.PadRight(20);
    Console.WriteLine("|" + str + "|");

    // Trim spaces.
    str = str.Trim();
    Console.WriteLine("|" + str + "|");

    // Pad on left with #s.
    str = str.PadLeft(10, '#');
    Console.WriteLine("|" + str + "|");

    // Pad on right with #s.
    str = str.PadRight(20, '#');
    Console.WriteLine("|" + str + "|");

    // Trim #s.
    str = str.Trim('#');
    Console.WriteLine("|" + str + "|");
  }
}
```

The output is shown here:

```
Original string: test
|      test|
|      test          |
|test|
|######test|
|######test##########|
|test|
```

Inserting, Removing, and Replacing

You can insert a string into another using the **Insert( )** method, shown here:

public string Insert(int *start*, string *str*)

Here, *str* is inserted into the invoking string at the index specified by *start*. The resulting string is returned.

You can remove a portion of a string using **Remove( )**, shown next:

public string Remove(int *start*)
public string Remove(int *start*, int *count*)

The first form begins at the index specified by *start* and removes all remaining characters in the string. The second form begins at count and removes *count* number of characters. In both cases, the resulting string is returned.

You can replace a portion of a string by using **Replace( )**. It has these forms:

public string Replace(char *ch1*, char *ch2*)
public string Replace(string *str1*, string *str2*)

The first form replaces all occurrences of *ch1* in the invoking string with *ch2*. The second form replaces all occurrences of *str1* in the invoking string with *str2*. In both cases, the resulting string is returned.

Here is an example that demonstrates **Insert( )**, **Remove( )**, and **Replace( )**:

```
// Inserting, replacing, and removing.

using System;

class InsRepRevDemo {
  static void Main() {
    string str = "This test";

    Console.WriteLine("Original string: " + str);

    // Insert
    str = str.Insert(5, "is a ");
    Console.WriteLine(str);

    // Replace string
    str = str.Replace("is", "was");
    Console.WriteLine(str);

    // Replace characters
    str = str.Replace('a', 'X');
    Console.WriteLine(str);

    // Remove
    str = str.Remove(4, 5);
    Console.WriteLine(str);
  }
}
```

The output is shown here:

```
Original string: This test
This is a test
Thwas was a test
ThwXs wXs X test
ThwX X test
```

Changing Case

String offers two convenient methods that enable you to change the case of letters within a string. These are called **ToUpper( )** and **ToLower( )**. Here are their simplest forms:

```
public string ToLower( )
public string ToUpper( )
```

ToLower( ) lowercases all letters within the invoking string. **ToUpper( )** uppercases all letters within the invoking string. The resulting string is returned. There are also versions of these methods that allow you to specify cultural settings.

Also available are the methods **ToUpperInvariant( )** and **ToLowerInvariant( )**, shown here:

```
public string ToUpperInvariant( )
public string ToLowerInvariant( )
```

These work like **ToUpper( )** and **ToLower( )** except that they use the invariant culture to perform the transformations to upper- or lowercase.

Using the Substring( ) Method

You can obtain a portion of a string by using the **Substring( )** method. It has these two forms:

public string Substring(int *idx*)
public string Substring(int *idx*, int *count*)

In the first form, the substring begins at the index specified by *idx* and runs to the end of the invoking string. In the second form, the substring begins at *idx* and runs for *count* characters. In each case, the substring is returned.

The following program demonstrates the **Substring( )** method:

```
// Use Substring().

using System;

class SubstringDemo {
  static void Main() {
    string str = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    Console.WriteLine("str: " + str);

    Console.Write("str.Substring(15): ");
    string substr = str.Substring(15);
    Console.WriteLine(substr);

    Console.Write("str.Substring(0, 15): ");
    substr = str.Substring(0, 15);
    Console.WriteLine(substr);
  }
}
```

The following output is produced:

```
str: ABCDEFGHIJKLMNOPQRSTUVWXYZ
str.Substring(15): PQRSTUVWXYZ
str.Substring(0, 15): ABCDEFGHIJKLMNO
```

The String Extension Methods

As mentioned earlier, **String** implements **IEnumerable<T>**. This means that beginning with C# 3.0, a **String** object can call the extension methods defined by **Enumerable** and **Queryable**, which are both in the **System.Linq** namespace. These extension methods primarily provide support for LINQ, but some can also be used for other purposes, such as certain types of string handling. See Chapter 19 for a discussion of extension methods.

Formatting

When a human-readable form of a built-in type, such as **int** or **double**, is needed, a string representation must be created. Although C# automatically supplies a default format for this representation, it is also possible to specify a format of your own choosing. For example, as you saw in Part I, it is possible to output numeric data using a dollars and cents format. A number

of methods format data, including **Console.WriteLine( )**, **String.Format( )**, and the **ToString( )** method defined for the numeric structure types. The same approach to formatting is used by all three; once you have learned to format data for one, you can apply it to the others.

Formatting Overview

Formatting is governed by two components: *format specifiers* and *format providers.* The form that the string representation of a value will take is controlled through the use of a format specifier. Thus, it is the format specifier that dictates how the human-readable form of the data will look. For example, to output a numeric value using scientific notation, you will use the E format specifier.

In many cases, the precise format of a value will be affected by the culture and language in which the program is running. For example, in the United States, money is represented in dollars. In Europe, money is represented in euros. To handle the cultural and language differences, C# uses format providers. A format provider defines the way that a format specifier will be interpreted. A format provider is created by implementing the **IFormatProvider** interface, which defines the **GetFormat( )** method. Format providers are predefined for the built-in numeric types and many other types in the .NET Framework. In general, you can format data without having to worry about specifying a format provider, and format providers are not examined further in this book.

To format data, include a format specifier in a call to a method that supports formatting. The use of format specifiers was introduced in Chapter 3, but is worthwhile reviewing here. The discussion that follows uses **Console.WriteLine( )**, but the same basic approach applies to other methods that support formatting.

To format data using **WriteLine( )**, use the version of **WriteLine( )** shown here:

WriteLine("*format string*", *arg0*, *arg1*, ... , *argN*);

In this version, the arguments to **WriteLine( )** are separated by commas and not **+** signs. The *format string* contains two items: regular, printing characters that are displayed as-is, and format commands.

Format commands take this general form:

{*argnum*, *width*: *fmt*}

Here, *argnum* specifies the number of the argument (starting from zero) to display. The minimum width of the field is specified by *width,* and the format specifier is represented by *fmt.* Both *width* and *fmt* are optional. Thus, in its simplest form, a format command simply indicates which argument to display. For example, **{0}** indicates *arg0*, **{1}** specifies *arg1,* and so on.

During execution, when a format command is encountered in the format string, the corresponding argument, as specified by *argnum,* is substituted and displayed. Thus, it is the position of a format specifier within the format string that determines where its matching data will be displayed. It is the argument number that determines which argument will be formatted.

If *fmt* is present, then the data is displayed using the specified format. Otherwise, the default format is used. If *width* is present, then output is padded with spaces to ensure that the minimum field width is attained. If *width* is positive, output is right-justified. If *width* is negative, output is left-justified.

The remainder of this chapter examines formatting and format specifiers in detail.

The Numeric Format Specifiers

There are several format specifiers defined for numeric data. They are shown in Table 22-4. Each format specifier can include an optional precision specifier. For example, to specify that a value be represented as a fixed-point value with two decimal places, use F2.

As explained, the precise effect of certain format specifiers depends upon the cultural settings. For example, the currency specifier, C, automatically displays a value in the monetary format of the selected culture. For most users, the default cultural information matches their locale and language. Thus, the same format specifier can be used without concern about the cultural context in which the program is executed.

Here is a program that demonstrates several of the numeric format specifiers:

```
// Demonstrate various format specifiers.

using System;

class FormatDemo {
  static void Main() {
    double v = 17688.65849;
    double v2 = 0.15;
    int x = 21;

    Console.WriteLine("{0:F2}", v);

    Console.WriteLine("{0:N5}", v);

    Console.WriteLine("{0:e}", v);

    Console.WriteLine("{0:r}", v);

    Console.WriteLine("{0:p}", v2);

    Console.WriteLine("{0:X}", x);

    Console.WriteLine("{0:D12}", x);

    Console.WriteLine("{0:C}", 189.99);
  }
}
```

The output is shown here:

```
17688.66
17,688.65849
1.768866e+004
17688.65849
15.00 %
15
000000000021
$189.99
```

Notice the effect of the precision specifier in several of the formats.

Specifier	Format	Meaning of Precision Specifier
C	Currency (that is, a monetary value).	Specifies the number of decimal places.
c	Same as C.	
D	Whole number numeric data. (Use with integers only.)	Minimum number of digits. Leading zeros will be used to pad the result, if necessary.
d	Same as D.	
E	Scientific notation (uses uppercase E).	Specifies the number of decimal places. The default is six.
e	Scientific notation (uses lowercase e).	Specifies the number of decimal places. The default is six.
F	Fixed-point notation.	Specifies the number of decimal places.
f	Same as F.	
G	Use either E or F format, whichever is shorter.	See E and F.
g	Use either e or f format, whichever is shorter.	See e and f.
N	Fixed-point notation, with comma separators.	Specifies the number of decimal places.
n	Same as N.	
P	Percentage	Specifies the number of decimal places.
p	Same as P.	
R or r	Numeric value that can be parsed, using **Parse( )**, back into its equivalent internal form. (This is called the "round-trip" format.)	Not used.
X	Hexadecimal (uses uppercase letters *A* through *F*).	Minimum number of digits. Leading zeros will be used to pad the result, if necessary.
x	Hexadecimal (uses lowercase letters *a* through *f*).	Minimum number of digits. Leading zeros will be used to pad the result if necessary.

TABLE 22-4 The Format Specifiers

Understanding Argument Numbers

It is important to understand that the argument associated with a format specifier is determined by the argument number, not the argument's position in the argument list. This means the same argument can be output more than once within the same call to **WriteLine( )**. It also

means that arguments can be displayed in a sequence different than they are specified in the argument list. For example, consider the following program:

```
using System;

class FormatDemo2 {
  static void Main() {

    // Format the same argument three different ways:
    Console.WriteLine("{0:F2}  {0:F3}  {0:e}", 10.12345);

    // Display arguments in non-sequential order.
    Console.WriteLine("{2:d} {0:d} {1:d}", 1, 2, 3);
  }
}
```

The output is shown here:

```
10.12  10.123  1.012345e+001
3 1 2
```

In the first **WriteLine( )** statement, the same argument, 10.12345, is formatted three different ways. This is possible because each format specifier refers to the first (and only) argument. In the second **WriteLine( )** statement, the three arguments are displayed in non-sequential order. Remember, there is no rule that format specifiers must use the arguments in sequence. Any format specifier can refer to any argument.

Using String.Format( ) and ToString( ) to Format Data

Although embedding format commands into **WriteLine( )** is a convenient way to format output, sometimes you will want to create a string that contains the formatted data, but not immediately display that string. Doing so lets you format data in advance, allowing you to output it later, to the device of your choosing. This is especially useful in a GUI environment, such as Windows, in which console-based I/O is rarely used, or for preparing output for a web page.

In general, there are two ways to obtain the formatted string representation of a value. One way is to use **String.Format( )**. The other is to pass a format specifier to the **ToString( )** method of the built-in numeric types. Each approach is examined here.

Using String.Format( ) to Format Values

You can obtain a formatted value by calling one of the **Format( )** methods defined by **String**. They are shown in Table 22-5. **Format( )** works much like **WriteLine( )**, except that it returns a formatted string rather than outputting it to the console.

Method	Description
public static string Format(string *str*, object *v*)	Formats *v* according to the first format command in *str.* Returns a copy of *str* in which formatted data has been substituted for the format command.
public static string Format(string *str*, object *v1*, object *v2*)	Formats *v1* according to the first format command in *str,* and *v2* according to the second format command in *str.* Returns a copy of *str* in which formatted data has been substituted for the format commands.
public static string Format(string *str*, object *v1,* object *v2*, object *v3*)	Formats *v1, v2,* and *v3* according to the corresponding format commands in *str.* Returns a copy of *str* in which formatted data has been substituted for the format commands.
public static string Format(string *str*, params object[ ] *v*)	Formats the values passed in *v* according to the format commands in *str.* Returns a copy of *str* in which formatted data has been substituted for each format command.
public static string Format(IFormatProvider *fmtprvdr*, string *str*, params object[ ] *v*)	Formats the values passed in *v* according to the format commands in *str* using the format provider specified by *fmtprvdr.* Returns a copy of *str* in which formatted data has been substituted for each format command.

TABLE 22-5 The **Format( )** Methods

Here is the previous format demonstration program rewritten to use **String.Format( )**. It produces the same output as the earlier version.

```
// Use String.Format() to format a value.

using System;

class FormatDemo {
  static void Main() {
    double v = 17688.65849;
    double v2 = 0.15;
    int x = 21;

    string str = String.Format("{0:F2}", v);
    Console.WriteLine(str);

    str = String.Format("{0:N5}", v);
    Console.WriteLine(str);
```

```
    str = String.Format("{0:e}", v);
    Console.WriteLine(str);

    str = String.Format("{0:r}", v);
    Console.WriteLine(str);

    str = String.Format("{0:p}", v2);
    Console.WriteLine(str);

    str = String.Format("{0:X}", x);
    Console.WriteLine(str);

    str = String.Format("{0:D12}", x);
    Console.WriteLine(str);

    str = String.Format("{0:C}", 189.99);
    Console.WriteLine(str);
  }
}
```

Like **WriteLine( )**, **String.Format( )** lets you embed regular text along with format specifiers, and you can use more than one format specifier and value. For example, consider this program, which displays the running sum and product of the numbers 1 through 10:

```
// A closer look at Format().

using System;

class FormatDemo2 {
  static void Main() {
    int i;
    int sum = 0;
    int prod = 1;
    string str;

    /* Display the running sum and product
       for the numbers 1 through 10. */
    for(i=1; i <= 10; i++) {
      sum += i;
      prod *= i;
      str = String.Format("Sum:{0,3:D}  Product:{1,8:D}",
                          sum, prod);
      Console.WriteLine(str);
    }
  }
}
```

The output is shown here:

```
Sum:  1  Product:       1
Sum:  3  Product:       2
Sum:  6  Product:       6
Sum: 10  Product:      24
Sum: 15  Product:     120
Sum: 21  Product:     720
Sum: 28  Product:    5040
```

```
Sum: 36  Product:    40320
Sum: 45  Product:   362880
Sum: 55  Product: 3628800
```

In the program, pay close attention to this statement:

```
str = String.Format("Sum:{0,3:D}  Product:{1,8:D}",
                    sum, prod);
```

This call to **Format( )** contains two format specifiers, one for **sum** and one for **prod**. Notice that the argument numbers are specified just as they are when using **WriteLine( )**. Also, notice that regular text, such as "Sum:" is included. This text is passed through and becomes part of the output string.

Using ToString( ) to Format Data

For all of the built-in numeric structure types, such as **Int32** or **Double**, you can use **ToString( )** to obtain a formatted string representation of the value. To do so, you will use this version of **ToString( )**:

public string ToString(string *fmt*)

It returns the string representation of the invoking object as specified by the format specifier passed in *fmt*. For example, the following statement creates a monetary representation of the value 188.99 through the use of the C format specifier:

```
string str = 189.99.ToString("C");
```

Notice how the format specifier is passed directly to **ToString( )**. Unlike embedded format commands used by **WriteLine( )** or **Format( )**, which supply an argument-number and field-width component, **ToString( )** requires only the format specifier, itself.

Here is a rewrite of the previous format program that uses **ToString( )** to obtain formatted strings. It produces the same output as the earlier versions.

```
// Use ToString() to format values.

using System;

class ToStringDemo {
  static void Main() {
    double v = 17688.65849;
    double v2 = 0.15;
    int x = 21;

    string str = v.ToString("F2");
    Console.WriteLine(str);

    str = v.ToString("N5");
    Console.WriteLine(str);

    str = v.ToString("e");
    Console.WriteLine(str);

    str = v.ToString("r");
```

```
    Console.WriteLine(str);

    str = v2.ToString("p");
    Console.WriteLine(str);

    str = x.ToString("X");
    Console.WriteLine(str);

    str = x.ToString("D12");
    Console.WriteLine(str);

    str = 189.99.ToString("C");
    Console.WriteLine(str);
  }
}
```

Creating a Custom Numeric Format

Although the predefined numeric format specifiers are quite useful, C# gives you the ability to define your own, custom format using a feature sometimes called *picture format.* The term *picture format* comes from the fact that you create a custom format by specifying an example (that is, picture) of how you want the output to look. This approach was mentioned briefly in Part I. Here it is examined in detail.

The Custom Format Placeholder Characters

When you create a custom format, you specify that format by creating an example (or picture) of what you want the data to look like. To do this, you use the characters shown in Table 22-6 as placeholders. Each is examined in turn.

The period specifies where the decimal point will be located.

The # placeholder specifies a digit position. The # can occur on the left or right side of the decimal point or by itself. When one or more #s occur on the right side of the decimal point, they specify the number of decimal digits to display. The value is rounded if necessary. When the # occurs to the left of the decimal point, it specifies the digit positions for the whole-number part of the value. Leading zeros will be added if necessary. If the whole-number portion of the value has more digits than there are #s, the entire whole-number portion will be displayed. In no cases will the whole-number portion of a value be truncated. If there is no decimal point, then the # causes the value to be rounded to its integer value. A zero value that is not significant, such as a trailing zero, will not be displayed. This causes a somewhat odd quirk, however, because a format such as #.## displays nothing at all if the value being formatted is zero. To output a zero value, use the 0 placeholder described next.

The 0 placeholder causes a leading or trailing 0 to be added to ensure that a minimum number of digits will be present. It can be used on both the right and left side of the decimal point. For example,

```
Console.WriteLine("{0:00##.#00}", 21.3);
```

displays this output:

```
0021.300
```

Values containing more digits will be displayed in full on the left side of the decimal point and rounded on the right side.

You can insert commas into large numbers by specifying a pattern that embeds a comma within a sequence of #s. For example, this:

```
Console.WriteLine("{0:#,###.#}", 3421.3);
```

displays

```
3,421.3.
```

It is not necessary to specify each comma for each position. Specifying one comma causes it to be inserted into the value every third digit from the left of the decimal point. For example,

```
Console.WriteLine("{0:#,###.#}", 8763421.3);
```

produces this output:

```
8,763,421.3.
```

Commas have a second meaning. When they occur on the immediate left of the decimal point, they act as a scaling factor. Each comma causes the value to be divided by 1,000. For example,

```
Console.WriteLine("Value in thousands: {0:#,###,.#}", 8763421.3);
```

produces this output:

```
Value in thousands: 8,763.4
```

As the output shows, the value is scaled in terms of thousands.

In addition to the placeholders, a custom format specifier can contain other characters. Any other characters are simply passed through, appearing in the formatted string exactly as they appear in the format specifier. For example, this **WriteLine( )** statement:

```
Console.WriteLine("Fuel efficiency is {0:##.# mpg}", 21.3);
```

produces this output:

```
Fuel efficiency is 21.3 mpg
```

You can also use the escape sequences, such as **\t** or **\n**, if necessary.

The *E* and *e* placeholders cause a value to be displayed in scientific notation. At least one 0, but possibly more, must follow the *E* or *e*. The 0s indicate the number of decimal digits that will be displayed. The decimal component will be rounded to fit the format. Using an uppercase *E* causes an uppercase *E* to be displayed; using a lowercase *e* causes a lowercase *e* to be displayed. To ensure that a sign character precedes the exponent, use the *E*+ or *e*+ forms. To display a sign character for negative values only, use *E*, *e*, *E*-, or *e*-.

The ";" is a separator that enables you to specify different formats for positive, negative, and zero values. Here is the general form of a custom format specifier that uses the ";":

positive-fmt;*negative-fmt*;*zero-fmt*

Here is an example:

```
Console.WriteLine("{0:#.##;(#.##);0.00}", num);
```

If **num** is positive, the value is displayed with two decimal places. If **num** is negative, the value is displayed with two decimal places and is between a set of parentheses. If **num** is zero, the string 0.00 is displayed. When using the separators, you don't need to supply all parts. If you just want to specify how positive and negative values will look, omit the zero format. To use the default for negative values, omit the negative format. In this case, the positive format and the zero format will be separated by two semicolons.

The following program demonstrates just a few of the many possible custom formats that you can create:

```
// Using custom formats.

using System;

class PictureFormatDemo {
  static void Main() {
    double num = 64354.2345;

    Console.WriteLine("Default format: " + num);

    // Display with 2 decimal places.
    Console.WriteLine("Value with two decimal places: " +
                      "{0:#.##}", num);

    // Display with commas and 2 decimal places.
    Console.WriteLine("Add commas: {0:#,###.##}", num);

    // Display using scientific notation.
    Console.WriteLine("Use scientific notation: " +
                      "{0:#.###e+00}", num);

    // Scale the value by 1000.
    Console.WriteLine("Value in 1,000s: " +
                      "{0:#0,}", num);

    /* Display positive, negative, and zero
       values differently. */
    Console.WriteLine("Display positive, negative, " +
                      "and zero values differently.");
    Console.WriteLine("{0:#.#;(#.##);0.00}", num);
    num = -num;
    Console.WriteLine("{0:#.##;(#.##);0.00}", num);
    num = 0.0;
    Console.WriteLine("{0:#.##;(#.##);0.00}", num);

    // Display a percentage.
    num = 0.17;
    Console.WriteLine("Display a percentage: {0:#%}", num);
  }
}
```

Placeholder	Meaning
#	Digit
.	Decimal point
,	Thousands separator
%	Percentage, which is the value being formatted multiplied by 100
0	Pads with leading and trailing zeros
;	Separates sections that describe the format for positive, negative, and zero values
E0 E+0 E-0 e0 e+0 e-0	Scientific notation

TABLE 22-6 Custom Format Placeholder Characters

The output is shown here:

```
Default format: 64354.2345
Value with two decimal places: 64354.23
Add commas: 64,354.23
Use scientific notation: 6.435e+04
Value in 1,000s: 64
Display positive, negative, and zero values differently.
64354.2
(64354.23)
0.00
Display a percentage: 17%
```

Formatting Date and Time

In addition to formatting numeric values, another data type to which formatting is often applied is **DateTime**. **DateTime** is a structure that represents date and time. Date and time values can be displayed a variety of ways. Here are just a few examples:

06/05/2006
Monday, June 5, 2006
12:59:00
12:59:00 PM

Also, the date and time representations can vary from country to country. For these reasons, the .NET Framework provides an extensive formatting subsystem for time and date values.

Date and time formatting is handled through format specifiers. The format specifiers for date and time are shown in Table 22-7. Because the specific date and time representation may vary from country to country and by language, the precise representation generated will be influenced by the cultural settings.

Specifier	Format
D	Date in long form.
d	Date in short form.
F	Date and time in long form.
f	Date and time in short form.
G	Date in short form, time in long form.
gg	Date in short form, time in short form.
M	Month and day.
m	Same as M.
O	A form of date and time that includes the time zone. The string produced by the O format can be parsed back into the equivalent date and time. This is called the "round trip" format.
o	Same as O.
R	Date and time in standard, GMT form.
r	Same as R.
s	A sortable form of date and time.
T	Time in long form.
t	Time in short form.
U	Long form, universal form of date and time. Time is displayed as UTC.
u	Short form, universal form of date and time.
Y	Month and year.
y	Same as Y.

TABLE 22-7 The Date and Time Format Specifiers

Here is a program that demonstrates the date and time format specifiers:

```
// Format time and date information.

using System;

class TimeAndDateFormatDemo {
  static void Main() {
    DateTime dt = DateTime.Now; // obtain current time

    Console.WriteLine("d format: {0:d}", dt);
    Console.WriteLine("D format: {0:D}", dt);

    Console.WriteLine("t format: {0:t}", dt);
    Console.WriteLine("T format: {0:T}", dt);

    Console.WriteLine("f format: {0:f}", dt);
    Console.WriteLine("F format: {0:F}", dt);
```

```
    Console.WriteLine("g format: {0:g}", dt);
    Console.WriteLine("G format: {0:G}", dt);

    Console.WriteLine("m format: {0:m}", dt);
    Console.WriteLine("M format: {0:M}", dt);

    Console.WriteLine("o format: {0:o}", dt);
    Console.WriteLine("O format: {0:O}", dt);

    Console.WriteLine("r format: {0:r}", dt);
    Console.WriteLine("R format: {0:R}", dt);

    Console.WriteLine("s format: {0:s}", dt);

    Console.WriteLine("u format: {0:u}", dt);
    Console.WriteLine("U format: {0:U}", dt);

    Console.WriteLine("y format: {0:y}", dt);
    Console.WriteLine("Y format: {0:Y}", dt);
  }
}
```

Sample output is shown here:

```
d format: 6/18/2008
D format: Wednesday, June 18, 2008
t format: 11:53 AM
T format: 11:53:09 AM
f format: Wednesday, June 18, 2008 11:53 AM
F format: Wednesday, June 18, 2008 11:53:09 AM
g format: 6/18/2008 11:53 AM
G format: 6/18/2008 11:53:09 AM
m format: June 18
M format: June 18
o format: 2008-06-18T11:53:09.5074933-05:00
O format: 2008-06-18T11:53:09.5074933-05:00
r format: Wed, 18 Jun 2008 11:53:09 GMT
R format: Wed, 18 Jun 2008 11:53:09 GMT
s format: 2008-06-18T11:53:09
u format: 2008-06-18 11:53:09Z
U format: Wednesday, June 18, 2008 4:53:09 PM
y format: June, 2008
Y format: June, 2008
```

The next program creates a very simple clock. The time is updated once every second. At the top of each hour, the computer's bell is sounded. It uses the **ToString( )** method of **DateTime** to obtain the formatted time prior to outputting it. If the top of the hour has been reached, then the alert character (**\a**) is appended to the formatted time, thus ringing the bell.

```
// A simple clock.

using System;

class SimpleClock {
  static void Main() {
    string t;
```

```
    int seconds;

    DateTime dt = DateTime.Now;
    seconds = dt.Second;

    for(;;) {
      dt = DateTime.Now;

      // update time if seconds change
      if(seconds != dt.Second) {
        seconds = dt.Second;

        t = dt.ToString("T");

        if(dt.Minute==0 && dt.Second==0)
          t = t + "\a"; // ring bell at top of hour

        Console.WriteLine(t);
      }
    }
  }
}
```

Creating a Custom Date and Time Format

Although the standard date and time format specifiers will apply to the vast majority of situations, you can create your own, custom formats. The process is similar to creating custom formats for the numeric types, as described earlier. In essence, you simply create an example (picture) of what you want the date and time information to look like. To create a custom date and time format, you will use one or more of the placeholders shown in Table 22-8.

If you examine Table 22-8, you will see that the placeholders *d, f, g, m, M, s,* and *t* are the same as the date and time format specifiers shown in Table 22-7. In general, if one of these characters is used by itself, it is interpreted as a format specifier. Otherwise, it is assumed to be a placeholder. If you want use one of these characters by itself but have it interpreted as a placeholder, then precede the character with a %.

The following program demonstrates several custom time and date formats:

```
// Format time and date information.

using System;

class CustomTimeAndDateFormatsDemo {
  static void Main() {
    DateTime dt = DateTime.Now;

    Console.WriteLine("Time is {0:hh:mm tt}", dt);
    Console.WriteLine("24 hour time is {0:HH:mm}", dt);
    Console.WriteLine("Date is {0:ddd MMM dd, yyyy}", dt);

    Console.WriteLine("Era: {0:gg}", dt);

    Console.WriteLine("Time with seconds: " +
                      "{0:HH:mm:ss tt}", dt);
```

```
        Console.WriteLine("Use m for day of month: {0:m}", dt);
        Console.WriteLine("use m for minutes: {0:%m}", dt);
    }
}
```

Placeholder	Replaced By
d	Day of month as a number between 1 and 31.
dd	Day of month as a number between 1 and 31. A leading 0 prefixes the values 1 through 9.
ddd	Abbreviated weekday name.
dddd	Full weekday name.
f, ff, fff, ffff, fffff, ffffff, fffffff	Fractional seconds, with the number of decimal places specified by the number of fs. (If uppercase Fs are used, trailing 0s are not displayed.)
g	Era.
h	Hour as a number between 1 and 12.
hh	Hour as a number between 1 and 12. A leading 0 prefixes the values 1 through 9.
H	Hour as a number between 0 and 23.
HH	Hour as a number between 0 and 23. A leading 0 prefixes the values 0 through 9.
K	Time zone offset in hours. It uses the value of the **DateTime.Kind** property to automatically adjust for local time and UTC time. (This specifier is now recommended over the z-based specifiers.)
m	Minutes.
mm	Minutes. A leading 0 prefixes the values 0 through 9.
M	Month as a number between 1 and 12.
MM	Month as a number between 1 and 12. A leading 0 prefixes the values 1 through 9.
MMM	Abbreviated month name.
MMMM	Full month name.
s	Seconds.
ss	Seconds. A leading 0 prefixes the values 0 through 9.
t	A or P, indicating A.M. or P.M.
tt	A.M. or P.M.
y	Year as two digits, unless only one digit is needed.
yy	Year as two digits. A leading 0 prefixes the values 0 through 9.
yyy	Year as three digits.
yyyy	Year using four digits.
yyyyy	Year using five digits.
z	Time zone offset in hours.

TABLE 22-8 The Custom Date and Time Placeholder Characters

Placeholder	Replaced By
zz	Time zone offset in hours. A leading 0 prefixes the values 0 through 9.
zzz	Time zone offset in hours and minutes.
:	Separator for time components.
/	Separator for date components.
%*fmt*	The standard format associated with *fmt*.

TABLE 22-8 The Custom Date and Time Placeholder Characters *(continued)*

The output is shown here:

```
Time is 11:55 AM
24 hour time is 11:55
Date is Wed Jun 18, 2008
Era: A.D.
Time with seconds: 11:55:52 AM
Use m for day of month: June 18
use m for minutes: 55
```

Formatting Enumerations

C# allows you to format the values defined by an enumeration. In general, enumeration values can be displayed using their name or their value. The enumeration format specifiers are shown in Table 22-9. Pay special attention to the G and F formats. Enumerations that will be used to represent bit-fields can be preceded by the **Flags** attribute. Typically, bit-fields hold values that represent individual bits and are arranged in powers of two. If the **Flags** attribute is present, then the G specifier will display the names of all of the values that comprise the value, assuming the value is valid. The F specifier will display the names of all of the values that comprise the value if the value can be constructed by ORing together two or more fields defined by the enumeration.

The following program demonstrates the enumeration specifiers:

```
// Format an enumeration.

using System;

class EnumFmtDemo {
  enum Direction { North, South, East, West }
  [Flags] enum Status { Ready=0x1, OffLine=0x2,
                        Waiting=0x4, TransmitOK=0x8,
                        ReceiveOK=0x10, OnLine=0x20 }

  static void Main() {
    Direction d = Direction.West;

    Console.WriteLine("{0:G}", d);
    Console.WriteLine("{0:F}", d);
```

```
    Console.WriteLine("{0:D}", d);
    Console.WriteLine("{0:X}", d);

    Status s = Status.Ready | Status.TransmitOK;

    Console.WriteLine("{0:G}", s);
    Console.WriteLine("{0:F}", s);
    Console.WriteLine("{0:D}", s);
    Console.WriteLine("{0:X}", s);
  }
}
```

The output is shown here:

```
West
West
3
00000003
Ready, TransmitOK
Ready, TransmitOK
9
00000009
```

Specifier	Meaning
D	Displays the value as a decimal integer.
d	Same as D.
F	Displays the name of the value. However, if the value can be created by ORing together two or more values defined by the enumeration, then the names of each part of the value will be displayed. This applies whether or not the **Flags** attribute has been specified.
f	Same as F.
G	Displays the name of the value. If the enumeration is preceded by the **Flags** attribute, then all names that are part of the value will be displayed (assuming a valid value).
g	Same as G.
X	Displays the value as a hexadecimal integer. Leading zeros will be added to ensure that at least eight digits are shown.
x	Same as X.

Table 22-9 The Enumeration Format Specifiers

This page intentionally left blank

CHAPTER 23

Multithreaded Programming

Although C# contains many exciting features, one of its most powerful is its built-in support for *multithreaded programming*. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a *thread*, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

Multithreaded programming relies on a combination of features defined by the C# language and by classes in the .NET Framework. Because support for multithreading is built into C#, many of the problems associated with multithreading in other languages are minimized or eliminated.

Multithreading Fundamentals

There are two distinct types of multitasking: process-based and thread-based. It is important to understand the difference between the two. A *process* is, in essence, a program that is executing. Thus, *process-based multitasking* is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking allows you to run a word processor at the same time you are using a spreadsheet or browsing the Internet. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

A *thread* is a dispatchable unit of executable code. The name comes from the concept of a "thread of execution." In a *thread-based* multitasking environment, all processes have at least one thread, but they can have more. This means that a single program can perform two or more tasks at once. For instance, a text editor can be formatting text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

The differences between process-based and thread-based multitasking can be summarized like this: Process-based multitasking handles the concurrent execution of programs. Thread-based multitasking deals with the concurrent execution of pieces of the same program.

The principal advantage of multithreading is that it enables you to write very efficient programs because it lets you utilize the idle time that is present in most programs. As you probably know, most I/O devices, whether they be network ports, disk drives, or the keyboard, are much slower than the CPU. Thus, a program will often spend a majority of its execution time waiting to send or receive information to or from a device. By using multithreading, your program can execute another task during this idle time. For example,

while one part of your program is sending a file over the Internet, another part can be reading keyboard input, and still another can be buffering the next block of data to send.

A thread can be in one of several states. In general terms, it can be *running*. It can be *ready to run* as soon as it gets CPU time. A running thread can be *suspended*, which is a temporary halt to its execution. It can later be *resumed*. A thread can be *blocked* when waiting for a resource. A thread can be *terminated*, in which case its execution ends and cannot be resumed.

The .NET Framework defines two types of threads: *foreground* and *background*. By default, when you create a thread, it is a foreground thread, but you can change it to a background thread. The only difference between foreground and background threads is that a background thread will be automatically terminated when all foreground threads in its process have stopped.

Along with thread-based multitasking comes the need for a special type of feature called *synchronization*, which allows the execution of threads to be coordinated in certain well-defined ways. C# has a complete subsystem devoted to synchronization, and its key features are also described here.

All processes have at least one thread of execution, which is usually called the *main thread* because it is the one that is executed when your program begins. Thus, the main thread is the thread that all of the preceding example programs in the book have been using. From the main thread, you can create other threads.

C# and the .NET Framework support both process-based and thread-based multitasking. Thus, using C#, you can create and manage both processes and threads. However, little programming effort is required to start a new process because each process is largely separate from the next. Rather, it is C#'s support for multithreading that is important. Because support for multithreading is built in, C# makes it easier to construct high-performance, multithreaded programs than do some other languages.

The classes that support multithreaded programming are defined in the **System.Threading** namespace. Thus, you will usually include this statement at the start of any multithreaded program:

```
using System.Threading;
```

The Thread Class

The multithreading system is built upon the **Thread** class, which encapsulates a thread of execution. The **Thread** class is sealed, which means that it cannot be inherited. **Thread** defines several methods and properties that help manage threads. Throughout this chapter, several of its most commonly used members will be examined.

Creating and Starting a Thread

There are a number of ways to create and start a thread. This section describes the basic mechanism. Various options are described later in this chapter.

To create a thread, instantiate an object of type **Thread**, which is a class defined in **System.Threading**. The simplest **Thread** constructor is shown here:

public Thread(ThreadStart *entryPoint*)

Here, *entryPoint* is the name of the method that will be called to begin execution of the thread. **ThreadStart** is a delegate defined by the .NET Framework as shown here:

public delegate void ThreadStart( )

Thus, your entry point method must have a **void** return type and take no arguments.

Once created, the new thread will not start running until you call its **Start( )** method, which is defined by **Thread**. The **Start( )** method has two forms. The one used here is

```
public void Start( )
```

Once started, the thread will run until the method specified by *entryPoint* returns. Thus, when *entryPoint* returns, the thread automatically stops. If you try to call **Start( )** on a thread that has already been started, a **ThreadStateException** will be thrown.

Here is an example that creates a new thread and starts it running:

```
// Create a thread of execution.

using System;
using System.Threading;

class MyThread {
  public int Count;
  string thrdName;

  public MyThread(string name) {
    Count = 0;
    thrdName = name;
  }

  // Entry point of thread.
  public void Run() {
    Console.WriteLine(thrdName + " starting.");

    do {
      Thread.Sleep(500);
      Console.WriteLine("In " + thrdName +
                        ", Count is " + Count);
      Count++;
    } while(Count < 10);

    Console.WriteLine(thrdName + " terminating.");
  }
}

class MultiThread {
  static void Main() {
    Console.WriteLine("Main thread starting.");

    // First, construct a MyThread object.
    MyThread mt = new MyThread("Child #1");

    // Next, construct a thread from that object.
    Thread newThrd = new Thread(mt.Run);

    // Finally, start execution of the thread.
    newThrd.Start();

    do {
      Console.Write(".");
```

```
      Thread.Sleep(100);
    } while (mt.Count != 10);

    Console.WriteLine("Main thread ending.");
  }
}
```

Let's look closely at this program. **MyThread** defines a class that will be used to create a second thread of execution. Inside its **Run( )** method, a loop is established that counts from 0 to 9. Notice the call to **Sleep( )**, which is a **static** method defined by **Thread**. The **Sleep( )** method causes the thread from which it is called to suspend execution for the specified period of milliseconds. When a thread suspends, another thread can run. The form used by the program is shown here:

public static void Sleep(int *milliseconds*)

The number of milliseconds to suspend is specified in *milliseconds*. If *milliseconds* is zero, the calling thread is suspended only to allow a waiting thread to execute.

Inside **Main( )**, a new **Thread** object is created by the following sequence of statements:

```
// First, construct a MyThread object.
MyThread mt = new MyThread("Child #1");

// Next, construct a thread from that object.
Thread newThrd = new Thread(mt.Run);

// Finally, start execution of the thread.
newThrd.Start();
```

As the comments suggest, first an object of **MyThread** is created. This object is then used to construct a **Thread** object by passing the **mt.Run( )** method as the entry point. Finally, execution of the new thread is started by calling **Start( )**. This causes **mt.Run( )** to begin executing in its own thread. After calling **Start( )**, execution of the main thread returns to **Main( )**, and it enters **Main( )**'s **do** loop. Both threads continue running, sharing the CPU, until their loops finish. The output produced by this program is as follows. (The precise output that you see may vary slightly because of differences in your execution environment, operating system, and task load.)

```
Main thread starting.
Child #1 starting.
.....In Child #1, Count is 0
.....In Child #1, Count is 1
.....In Child #1, Count is 2
.....In Child #1, Count is 3
.....In Child #1, Count is 4
.....In Child #1, Count is 5
.....In Child #1, Count is 6
.....In Child #1, Count is 7
.....In Child #1, Count is 8
.....In Child #1, Count is 9
Child #1 terminating.
Main thread ending.
```

Often in a multithreaded program, you will want the main thread to be the last thread to finish running. Technically, a program continues to run until all of its foreground threads

have finished. Thus, having the main thread finish last is not a requirement. It is, however, often good practice to follow because it clearly defines your program's endpoint. The preceding program tries to ensure that the main thread will finish last, by checking the value of **Count** within **Main( )**'s **do** loop, stopping when **Count** equals 10, and through the use of calls to **Sleep( )**. However, this is an imperfect approach. Later in this chapter, you will see better ways for one thread to wait until another finishes.

Some Simple Improvements

While the preceding program is perfectly valid, some easy improvements will make it more efficient. First, it is possible to have a thread begin execution as soon as it is created. In the case of **MyThread**, this is done by instantiating a **Thread** object inside **MyThread**'s constructor. Second, there is no need for **MyThread** to store the name of the thread since **Thread** defines a property called **Name** that can be used for this purpose. **Name** is defined like this:

```
public string Name { get; set; }
```

Since **Name** is a read-write property, you can use it to set the name of a thread or to retrieve the thread's name.

Here is a version of the preceding program that makes these three improvements:

```
// An alternate way to start a thread.

using System;
using System.Threading;

class MyThread {
  public int Count;
  public Thread Thrd;

  public MyThread(string name) {
    Count = 0;
    Thrd = new Thread(this.Run);
    Thrd.Name = name; // set the name of the thread
    Thrd.Start(); // start the thread
  }

  // Entry point of thread.
  void Run() {
    Console.WriteLine(Thrd.Name + " starting.");

    do {
      Thread.Sleep(500);
      Console.WriteLine("In " + Thrd.Name +
                        ", Count is " + Count);
      Count++;
    } while(Count < 10);

    Console.WriteLine(Thrd.Name + " terminating.");
  }
}

class MultiThreadImproved {
  static void Main() {
```

```
    Console.WriteLine("Main thread starting.");

    // First, construct a MyThread object.
    MyThread mt = new MyThread("Child #1");

    do {
      Console.Write(".");
      Thread.Sleep(100);
    } while (mt.Count != 10);

    Console.WriteLine("Main thread ending.");
  }
}
```

This version produces the same output as before. Notice that the thread object is stored in **Thrd** inside **MyThread**.

Creating Multiple Threads

The preceding examples have created only one child thread. However, your program can spawn as many threads as it needs. For example, the following program creates three child threads:

```
// Create multiple threads of execution.

using System;
using System.Threading;

class MyThread {
  public int Count;
  public Thread Thrd;

  public MyThread(string name) {
    Count = 0;
    Thrd = new Thread(this.Run);
    Thrd.Name = name;
    Thrd.Start();
  }

  // Entry point of thread.
  void Run() {
    Console.WriteLine(Thrd.Name + " starting.");

    do {
      Thread.Sleep(500);
      Console.WriteLine("In " + Thrd.Name +
                        ", Count is " + Count);
      Count++;
    } while(Count < 10);

    Console.WriteLine(Thrd.Name + " terminating.");
  }
}
```

```
class MoreThreads {
  static void Main() {
    Console.WriteLine("Main thread starting.");

    // Construct three threads.
    MyThread mt1 = new MyThread("Child #1");
    MyThread mt2 = new MyThread("Child #2");
    MyThread mt3 = new MyThread("Child #3");

    do {
      Console.Write(".");
      Thread.Sleep(100);
    } while (mt1.Count < 10 ||
             mt2.Count < 10 ||
             mt3.Count < 10);

    Console.WriteLine("Main thread ending.");
  }
}
```

Sample output from this program is shown next:

```
Main thread starting.
.Child #1 starting.
Child #2 starting.
Child #3 starting.
....In Child #1, Count is 0
In Child #2, Count is 0
In Child #3, Count is 0
.....In Child #1, Count is 1
In Child #2, Count is 1
In Child #3, Count is 1
.....In Child #1, Count is 2
In Child #2, Count is 2
In Child #3, Count is 2
.....In Child #1, Count is 3
In Child #2, Count is 3
In Child #3, Count is 3
.....In Child #1, Count is 4
In Child #2, Count is 4
In Child #3, Count is 4
.....In Child #1, Count is 5
In Child #2, Count is 5
In Child #3, Count is 5
.....In Child #1, Count is 6
In Child #2, Count is 6
In Child #3, Count is 6
.....In Child #1, Count is 7
In Child #2, Count is 7
In Child #3, Count is 7
.....In Child #1, Count is 8
In Child #2, Count is 8
In Child #3, Count is 8
.....In Child #1, Count is 9
```

```
Child #1 terminating.
In Child #2, Count is 9
Child #2 terminating.
In Child #3, Count is 9
Child #3 terminating.
Main thread ending.
```

As you can see, once started, all three child threads share the CPU. Again, because of differences among system configurations, operating systems, and other environmental factors, when you run the program, the output you see may differ slightly from that shown here.

Determining When a Thread Ends

Often it is useful to know when a thread has ended. In the preceding examples, this was attempted by watching the **Count** variable—hardly a satisfactory or generalizable solution. Fortunately, **Thread** provides two means by which you can determine whether a thread has ended. First, you can interrogate the read-only **IsAlive** property for the thread. It is defined like this:

```
public bool IsAlive { get; }
```

IsAlive returns true if the thread upon which it is called is still running. It returns false otherwise. To try **IsAlive**, substitute this version of **MoreThreads** for the one shown in the preceding program:

```
// Use IsAlive to wait for threads to end.
class MoreThreads {
  static void Main() {
    Console.WriteLine("Main thread starting.");

    // Construct three threads.
    MyThread mt1 = new MyThread("Child #1");
    MyThread mt2 = new MyThread("Child #2");
    MyThread mt3 = new MyThread("Child #3");

    do {
      Console.Write(".");
      Thread.Sleep(100);
    } while (mt1.Thrd.IsAlive &&
             mt2.Thrd.IsAlive &&
             mt3.Thrd.IsAlive);

    Console.WriteLine("Main thread ending.");
  }
}
```

This version produces the same output as before. The only difference is that it uses **IsAlive** to wait for the child threads to terminate.

Another way to wait for a thread to finish is to call **Join( )**. Its simplest form is shown here:

```
public void Join( )
```

Join( ) waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread *joins* it. A **ThreadStateException** will be thrown if the thread has not been started. Additional forms of **Join( )** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

Here is a program that uses **Join( )** to ensure that the main thread is the last to stop:

```
// Use Join().

using System;
using System.Threading;

class MyThread {
  public int Count;
  public Thread Thrd;

  public MyThread(string name) {
    Count = 0;
    Thrd = new Thread(this.Run);
    Thrd.Name = name;
    Thrd.Start();
  }

  // Entry point of thread.
  void Run() {
    Console.WriteLine(Thrd.Name + " starting.");

    do {
      Thread.Sleep(500);
      Console.WriteLine("In " + Thrd.Name +
                        ", Count is " + Count);
      Count++;
    } while(Count < 10);

    Console.WriteLine(Thrd.Name + " terminating.");
  }
}

// Use Join() to wait for threads to end.
class JoinThreads {
  static void Main() {
    Console.WriteLine("Main thread starting.");

    // Construct three threads.
    MyThread mt1 = new MyThread("Child #1");
    MyThread mt2 = new MyThread("Child #2");
    MyThread mt3 = new MyThread("Child #3");

    mt1.Thrd.Join();
    Console.WriteLine("Child #1 joined.");

    mt2.Thrd.Join();
    Console.WriteLine("Child #2 joined.");
```

```
    mt3.Thrd.Join();
    Console.WriteLine("Child #3 joined.");

    Console.WriteLine("Main thread ending.");
  }
}
```

Sample output from this program is shown here. Remember when you try the program, your output may vary slightly.

```
Main thread starting.
Child #1 starting.
Child #2 starting.
Child #3 starting.
In Child #1, Count is 0
In Child #2, Count is 0
In Child #3, Count is 0
In Child #1, Count is 1
In Child #2, Count is 1
In Child #3, Count is 1
In Child #1, Count is 2
In Child #2, Count is 2
In Child #3, Count is 2
In Child #1, Count is 3
In Child #2, Count is 3
In Child #3, Count is 3
In Child #1, Count is 4
In Child #2, Count is 4
In Child #3, Count is 4
In Child #1, Count is 5
In Child #2, Count is 5
In Child #3, Count is 5
In Child #1, Count is 6
In Child #2, Count is 6
In Child #3, Count is 6
In Child #1, Count is 7
In Child #2, Count is 7
In Child #3, Count is 7
In Child #1, Count is 8
In Child #2, Count is 8
In Child #3, Count is 8
In Child #1, Count is 9
Child #1 terminating.
In Child #2, Count is 9
Child #2 terminating.
In Child #3, Count is 9
Child #3 terminating.
Child #1 joined.
Child #2 joined.
Child #3 joined.
Main thread ending.
```

As you can see, after the calls to **Join( )** return, the threads have stopped executing.

Passing an Argument to a Thread

In the early days of the .NET Framework, it was not possible to pass an argument to a thread when the thread was started because the method that serves as the entry point to a thread could not have a parameter. If information needed to be passed to a thread, various workarounds (such as using a shared variable) were required. However, this deficiency was subsequently remedied, and today it is possible to pass an argument to a thread. To do so, you must use different forms of **Start( )**, the **Thread** constructor, and the entry point method.

An argument is passed to a thread through this version of **Start( )**:

public void Start(object *arg*)

The object passed to *arg* is automatically passed to the thread's entry point method. Thus, to pass an argument to a thread, you pass it to **Start( )**.

To make use of the parameterized version of **Start( )**, you must use the following form of the **Thread** constructor:

public Thread(ParameterizedThreadStart *entryPoint*)

Here, *entryPoint* is the name of the method that will be called to begin execution of the thread. Notice in this version, the type of *entryPoint* is **ParameterizedThreadStart** rather than **ThreadStart**, as used by the preceding examples. **ParameterizedThreadStart** is a delegate that is declared as shown here:

public delegate void ParameterizedThreadStart(object *arg*)

As you can see, this delegate takes an argument of type **object**. Therefore, to use this form of the **Thread** constructor, the thread entry point method must have an **object** parameter.

Here is an example that demonstrates the passing of an argument to a thread:

```
// Passing an argument to the thread method.

using System;
using System.Threading;

class MyThread {
  public int Count;
  public Thread Thrd;

  // Notice that MyThread is also passed an int value.
  public MyThread(string name, int num) {
    Count = 0;

    // Explicitly invoke ParameterizedThreadStart constructor
    // for the sake of illustration.
    Thrd = new Thread(this.Run);

    Thrd.Name = name;

     // Here, Start() is passed num as an argument.
    Thrd.Start(num);
  }
```

```
  // Notice that this version of Run() has
  // a parameter of type object.
  void Run(object num) {
    Console.WriteLine(Thrd.Name +
                      " starting with count of " + num);

    do {
      Thread.Sleep(500);
      Console.WriteLine("In " + Thrd.Name +
                        ", Count is " + Count);
      Count++;
    } while(Count < (int) num);

    Console.WriteLine(Thrd.Name + " terminating.");
  }
}

class PassArgDemo {
  static void Main() {

    // Notice that the iteration count is passed
    // to these two MyThread objects.
    MyThread mt = new MyThread("Child #1", 5);
    MyThread mt2 = new MyThread("Child #2", 3);

    do {
      Thread.Sleep(100);
    } while (mt.Thrd.IsAlive | mt2.Thrd.IsAlive);

    Console.WriteLine("Main thread ending.");
  }
}
```

The output is shown here. (The actual output you see may vary.)

```
Child #1 starting with count of 5
Child #2 starting with count of 3
In Child #2, Count is 0
In Child #1, Count is 0
In Child #1, Count is 1
In Child #2, Count is 1
In Child #2, Count is 2
Child #2 terminating.
In Child #1, Count is 2
In Child #1, Count is 3
In Child #1, Count is 4
Child #1 terminating.
Main thread ending.
```

As the output shows, the first thread iterates five times and the second thread iterates three times. The iteration count is specified in the **MyThread** constructor and then passed to the thread entry method **Run( )** through the use of the **ParameterizedThreadStart** version of **Start( )**.

The IsBackground Property

As mentioned earlier, the .NET Framework defines two types of threads: foreground and background. The only difference between the two is that a process won't end until all of its foreground threads have ended, but background threads are terminated automatically after all foreground threads have stopped. By default, a thread is created as a foreground thread. It can be changed to a background thread by using the **IsBackground** property defined by **Thread**, as shown here:

```
public bool IsBackground { get; set; }
```

To set a thread to background, simply assign **IsBackground** a **true** value. A value of **false** indicates a foreground thread.

Thread Priorities

Each thread has a priority setting associated with it. A thread's priority determines, in part, how frequently a thread gains access to the CPU. In general, low-priority threads gain access to the CPU less often than high-priority threads. As a result, within a given period of time, a low-priority thread will often receive less CPU time than a high-priority thread. As you might expect, how much CPU time a thread receives profoundly affects its execution characteristics and its interaction with other threads currently executing in the system.

It is important to understand that factors other than a thread's priority can also affect how frequently a thread gains access to the CPU. For example, if a high-priority thread is waiting on some resource, perhaps for keyboard input, it will be blocked, and a lower-priority thread will run. Thus, in this situation, a low-priority thread may gain greater access to the CPU than the high-priority thread over a specific period. Finally, precisely how task scheduling is implemented by the operating system affects how CPU time is allocated.

When a child thread is started, it receives a default priority setting. You can change a thread's priority through the **Priority** property, which is a member of **Thread**. This is its general form:

```
public ThreadPriority Priority{ get; set; }
```

ThreadPriority is an enumeration that defines the following five priority settings:

```
ThreadPriority.Highest
ThreadPriority.AboveNormal
ThreadPriority.Normal
ThreadPriority.BelowNormal
ThreadPriority.Lowest
```

The default priority setting for a thread is **ThreadPriority.Normal**.

To understand how priorities affect thread execution, we will use an example that executes two threads, one having a higher priority than the other. The threads are created as instances of the **MyThread** class. The **Run( )** method contains a loop that counts the number of iterations. The loop stops when either the count reaches 1,000,000,000 or the static variable **stop** is **true**. Initially, **stop** is set to **false**. The first thread to count to 1,000,000,000 sets **stop** to **true**. This causes the second thread to terminate with its next time slice. Each time through the loop, the string in **currentName** is checked against the name of the executing thread. If

they don't match, it means that a task-switch occurred. Each time a task-switch happens, the name of the new thread is displayed and **currentName** is given the name of the new thread. This allows you to watch how often each thread has access to the CPU. After both threads stop, the number of iterations for each loop is displayed.

```
// Demonstrate thread priorities.

using System;
using System.Threading;

class MyThread {
  public int Count;
  public Thread Thrd;

  static bool stop = false;
  static string currentName;

  /* Construct a new thread. Notice that this
     constructor does not actually start the
     threads running. */
  public MyThread(string name) {
    Count = 0;
    Thrd = new Thread(this.Run);
    Thrd.Name = name;
    currentName = name;
  }

  // Begin execution of new thread.
  void Run() {
    Console.WriteLine(Thrd.Name + " starting.");
    do {
      Count++;

      if(currentName != Thrd.Name) {
        currentName = Thrd.Name;
        Console.WriteLine("In " + currentName);
      }

    } while(stop == false && Count < 1000000000);
    stop = true;

    Console.WriteLine(Thrd.Name + " terminating.");
  }
}

class PriorityDemo {
  static void Main() {
    MyThread mt1 = new MyThread("High Priority");
    MyThread mt2 = new MyThread("Low Priority");

    // Set the priorities.
    mt1.Thrd.Priority = ThreadPriority.AboveNormal;
    mt2.Thrd.Priority = ThreadPriority.BelowNormal;
```

```
    // Start the threads.
    mt1.Thrd.Start();
    mt2.Thrd.Start();

    mt1.Thrd.Join();
    mt2.Thrd.Join();

    Console.WriteLine();
    Console.WriteLine(mt1.Thrd.Name + " thread counted to " +
                      mt1.Count);
    Console.WriteLine(mt2.Thrd.Name + " thread counted to " +
                      mt2.Count);
  }
}
```

Here is sample output:

```
High Priority starting.
In High Priority
Low Priority starting.
In Low Priority
In High Priority
In Low Priority
In High Priority
In Low Priority
In High Priority
In Low Priority
In High Priority
In Low Priority
In High Priority
High Priority terminating.
Low Priority terminating.

High Priority thread counted to 1000000000
Low Priority thread counted to 23996334
```

In this run, of the CPU time allotted to the program, the high-priority thread got approximately 98 percent. Of course, the precise output you see may vary, depending on the speed of your CPU and the number of other tasks running on the system. Which version of Windows you are running will also have an effect.

Because multithreaded code can behave differently in different environments, you should never base your code on the execution characteristics of a single environment. For example, in the preceding example, it would be a mistake to assume that the low-priority thread will always execute at least a small amount of time before the high-priority thread finishes. In a different environment, the high-priority thread might complete before the low-priority thread has executed even once, for example.

Synchronization

When using multiple threads, you will sometimes need to coordinate the activities of two or more of the threads. The process by which this is achieved is called *synchronization.* The most common reason for using synchronization is when two or more threads need access to a

shared resource that can be used by only one thread at a time. For example, when one thread is writing to a file, a second thread must be prevented from doing so at the same time. Another situation in which synchronization is needed is when one thread is waiting for an event that is caused by another thread. In this case, there must be some means by which the first thread is held in a suspended state until the event has occurred. Then the waiting thread must resume execution.

The key to synchronization is the concept of a *lock,* which controls access to a block of code within an object. When an object is locked by one thread, no other thread can gain access to the locked block of code. When the thread releases the lock, the object is available for use by another thread.

The lock feature is built into the C# language. Thus, all objects can be synchronized. Synchronization is supported by the keyword **lock**. Since synchronization was designed into C# from the start, it is much easier to use than you might first expect. In fact, for many programs, the synchronization of objects is almost transparent.

The general form of **lock** is shown here:

```
lock(lockObj) {
  // statements to be synchronized
}
```

Here, *lockObj* is a reference to the object being synchronized. If you want to synchronize only a single statement, the curly braces are not needed. A **lock** statement ensures that the section of code protected by the lock for the given object can be used only by the thread that obtains the lock. All other threads are blocked until the lock is removed. The lock is released when the block is exited.

The object you lock on is an object that represents the resource being synchronized. In some cases, this will be an instance of the resource itself or simply an arbitrary instance of **object** that is being used to provide synchronization. A key point to understand about **lock** is that the lock-on object should not be publically accessible. Why? Because it is possible that another piece of code that is outside your control could lock on the object and never release it. In the past, it was common to use a construct such as **lock(this)**. However, this works only if **this** refers to a private object. Because of the potential for error and conceptual mistakes in this regard, **lock(this)** is no longer recommended for general use. Instead, it is better to simply create a private object on which to lock. This is the approach used by the examples in this chapter. Be aware that you will still find many examples of **lock(this)** in legacy C# code. In some cases, it will be safe. In others, it will need to be changed to avoid problems.

The following program demonstrates synchronization by controlling access to a method called **SumIt( )**, which sums the elements of an integer array:

```
// Use lock to synchronize access to an object.

using System;
using System.Threading;

class SumArray {
  int sum;
  object lockOn = new object(); // a private object to lock on

  public int SumIt(int[] nums) {
```

```
    lock(lockOn) { // lock the entire method
      sum = 0; // reset sum

      for(int i=0; i < nums.Length; i++) {
        sum += nums[i];
        Console.WriteLine("Running total for " +
               Thread.CurrentThread.Name +
               " is " + sum);
        Thread.Sleep(10); // allow task-switch
      }
      return sum;
    }
  }
}

class MyThread {
  public Thread Thrd;
  int[] a;
  int answer;

  // Create one SumArray object for all instances of MyThread.
  static SumArray sa = new SumArray();

  // Construct a new thread.
  public MyThread(string name, int[] nums) {
    a = nums;
    Thrd = new Thread(this.Run);
    Thrd.Name = name;
    Thrd.Start(); // start the thread
  }

  // Begin execution of new thread.
  void Run() {
    Console.WriteLine(Thrd.Name + " starting.");

    answer = sa.SumIt(a);

    Console.WriteLine("Sum for " + Thrd.Name +
                      " is " + answer);

    Console.WriteLine(Thrd.Name + " terminating.");
  }
}

class Sync {
  static void Main() {
    int[] a = {1, 2, 3, 4, 5};

    MyThread mt1 = new MyThread("Child #1", a);
    MyThread mt2 = new MyThread("Child #2", a);

    mt1.Thrd.Join();
    mt2.Thrd.Join();
  }
}
```

Here is sample output from the program. (The actual output you see may vary slightly.)

```
Child #1 starting.
Running total for Child #1 is 1
Child #2 starting.
Running total for Child #1 is 3
Running total for Child #1 is 6
Running total for Child #1 is 10
Running total for Child #1 is 15
Running total for Child #2 is 1
Sum for Child #1 is 15
Child #1 terminating.
Running total for Child #2 is 3
Running total for Child #2 is 6
Running total for Child #2 is 10
Running total for Child #2 is 15
Sum for Child #2 is 15
Child #2 terminating.
```

As the output shows, both threads compute the proper sum of 15.

Let's examine this program in detail. The program creates three classes. The first is **SumArray**. It defines the method **SumIt( )**, which sums an integer array. The second class is **MyThread**, which uses a **static** object called **sa** that is of type **SumArray**. Thus, only one object of **SumArray** is shared by all objects of type **MyThread**. This object is used to obtain the sum of an integer array. Notice that **SumArray** stores the running total in a field called **sum**. Thus, if two threads use **SumIt( )** concurrently, both will be attempting to use **sum** to hold the running total. Because this will cause errors, access to **SumIt( )** must be synchronized. Finally, the class **Sync** creates two threads and has them compute the sum of an integer array.

Inside **SumIt( )**, the **lock** statement prevents simultaneous use of the method by different threads. Notice that **lock** uses **lockOn** as the object being synchronized. This is a private object that is used solely for synchronization. **Sleep( )** is called to purposely allow a task-switch to occur, if one can—but it can't in this case. Because the code within **SumIt( )** is locked, it can be used by only one thread at a time. Thus, when the second child thread begins execution, it does not enter **SumIt( )** until after the first child thread is done with it. This ensures the correct result is produced.

To understand the effects of **lock** fully, try removing it from the body of **SumIt( )**. After doing this, **SumIt( )** is no longer synchronized, and any number of threads can use it concurrently on the same object. The problem with this is that the running total is stored in **sum**, which will be changed by each thread that calls **SumIt( )**. Thus, when two threads call **SumIt( )** at the same time on the same object, incorrect results are produced because **sum** reflects the summation of both threads, mixed together. For example, here is sample output from the program after **lock** has been removed from **SumIt( )**:

```
Child #1 starting.
Running total for Child #1 is 1
Child #2 starting.
Running total for Child #2 is 1
Running total for Child #1 is 3
Running total for Child #2 is 5
Running total for Child #1 is 8
Running total for Child #2 is 11
```

```
Running total for Child #1 is 15
Running total for Child #2 is 19
Running total for Child #1 is 24
Running total for Child #2 is 29
Sum for Child #1 is 29
Child #1 terminating.
Sum for Child #2 is 29
Child #2 terminating.
```

As the output shows, both child threads are using **SumIt( )** at the same time on the same object, and the value of **sum** is corrupted.

The effects of **lock** are summarized here:

- For any given object, once a lock has been acquired, the object is locked and no other thread can acquire the lock.
- Other threads trying to acquire the lock on the same object will enter a wait state until the code is unlocked.
- When a thread leaves the locked block, the object is unlocked.

An Alternative Approach

Although locking a method's code, as shown in the previous example, is an easy and effective means of achieving synchronization, it will not work in all cases. For example, you might want to synchronize access to a method of a class you did not create, which is itself not synchronized. This can occur if you want to use a class that was written by a third party and for which you do not have access to the source code. Thus, it is not possible for you to add a **lock** statement to the appropriate method within the class. How can access to an object of this class be synchronized? Fortunately, the solution to this problem is simple: Lock access to the object from code outside the object by specifying the object in a **lock** statement. For example, here is an alternative implementation of the preceding program. Notice that the code within **SumIt( )** is no longer locked and no longer declares the **lockOn** object. Instead, calls to **SumIt( )** are locked within **MyThread**.

```
// Another way to use lock to synchronize access to an object.

using System;
using System.Threading;

class SumArray {
  int sum;

  public int SumIt(int[] nums) {
    sum = 0; // reset sum

    for(int i=0; i < nums.Length; i++) {
      sum += nums[i];
      Console.WriteLine("Running total for " +
             Thread.CurrentThread.Name +
             " is " + sum);
      Thread.Sleep(10); // allow task-switch
    }
    return sum;
```

```
  }
}

class MyThread {
  public Thread Thrd;
  int[] a;
  int answer;

  /* Create one SumArray object for all
     instances of MyThread. */
  static SumArray sa = new SumArray();

  // Construct a new thread.
  public MyThread(string name, int[] nums) {
    a = nums;
    Thrd = new Thread(this.Run);
    Thrd.Name = name;
    Thrd.Start(); // start the thread
  }

  // Begin execution of new thread.
  void Run() {
    Console.WriteLine(Thrd.Name + " starting.");

    // Lock calls to SumIt().
    lock(sa) answer = sa.SumIt(a);

    Console.WriteLine("Sum for " + Thrd.Name +
                      " is " + answer);

    Console.WriteLine(Thrd.Name + " terminating.");
  }
}

class Sync {
  static void Main() {
    int[] a = {1, 2, 3, 4, 5};

    MyThread mt1 = new MyThread("Child #1", a);
    MyThread mt2 = new MyThread("Child #2", a);

    mt1.Thrd.Join();
    mt2.Thrd.Join();
  }
}
```

Here, the call to **sa.SumIt( )** is locked, rather than the code inside **SumIt( )** itself. The code that accomplishes this is shown here:

```
// Lock calls to SumIt().
lock(sa) answer = sa.SumIt(a);
```

Because **sa** is a private object, it is safe to lock on. Using this approach, the program produces the same correct results as the original approach.

The Monitor Class and lock

The C# keyword **lock** is really just shorthand for using the synchronization features defined by the **Monitor** class, which is defined in the **System.Threading** namespace. **Monitor** defines several methods that control or manage synchronization. For example, to obtain a lock on an object, call **Enter( )**. To release a lock, call **Exit( )**. These methods are shown here:

```
public static void Enter(object syncOb)
public static void Exit(object syncOb)
```

Here, *syncOb* is the object being synchronized. If the object is not available when **Enter( )** is called, the calling thread will wait until it becomes available. You will seldom use **Enter( )** or **Exit( )**, however, because a **lock** block automatically provides the equivalent. For this reason, **lock** is the preferred method of obtaining a lock on an object when programming in C#.

One method in **Monitor** that you may find useful on occasion is **TryEnter( )**. One of its forms is shown here:

```
public static bool TryEnter(object syncOb)
```

It returns true if the calling thread obtains a lock on *syncOb* and false if it doesn't. In no case does the calling thread wait. You could use this method to implement an alternative if the desired object is unavailable.

Monitor also defines these three methods: **Wait( )**, **Pulse( )**, and **PulseAll( )**. They are described in the next section.

Thread Communication Using Wait( ), Pulse( ), and PulseAll( )

Consider the following situation. A thread called *T* is executing inside a **lock** block and needs access to a resource, called *R*, that is temporarily unavailable. What should *T* do? If *T* enters some form of polling loop that waits for *R*, then *T* ties up the object, blocking other threads' access to it. This is a less than optimal solution because it partially defeats the advantages of programming for a multithreaded environment. A better solution is to have *T* temporarily relinquish control of the object, allowing another thread to run. When *R* becomes available, *T* can be notified and resume execution. Such an approach relies upon some form of interthread communication in which one thread can notify another that it is blocked and be notified when it can resume execution. C# supports interthread communication with the **Wait( )**, **Pulse( )**, and **PulseAll( )** methods.

The **Wait( )**, **Pulse( )**, and **PulseAll( )** methods are defined by the **Monitor** class. These methods can be called only from within a locked block of code. Here is how they are used. When a thread is temporarily blocked from running, it calls **Wait( )**. This causes the thread to go to sleep and the lock for that object to be released, allowing another thread to use the object. At a later point, the sleeping thread is awakened when some other thread enters the same lock and calls **Pulse( )** or **PulseAll( )**. A call to **Pulse( )** resumes the first thread in the queue of threads waiting for the lock. A call to **PulseAll( )** signals the release of the lock to all waiting threads.

Here are two commonly used forms of **Wait( )**:

```
public static bool Wait(object waitOb)
public static bool Wait(object waitOb, int milliseconds)
```

The first form waits until notified. The second form waits until notified or until the specified period of milliseconds has expired. For both, *waitOb* specifies the object upon which to wait.

Here are the general forms for **Pulse( )** and **PulseAll( )**:

public static void Pulse(object *waitOb*)
public static void PulseAll(object *waitOb*)

Here, *waitOb* is the object being released.

A **SynchronizationLockException** will be thrown if **Wait( )**, **Pulse( )**, or **PulseAll( )** is called from code that is not within synchronized code, such as a **lock** block.

An Example That Uses Wait( ) and Pulse( )

To understand the need for and the application of **Wait( )** and **Pulse( )**, we will create a program that simulates the ticking of a clock by displaying the words "Tick" and "Tock" on the screen. To accomplish this, we will create a class called **TickTock** that contains two methods: **Tick( )** and **Tock( )**. The **Tick( )** method displays the word "Tick" and **Tock( )** displays "Tock". To run the clock, two threads are created, one that calls **Tick( )** and one that calls **Tock( )**. The goal is to make the two threads execute in a way that the output from the program displays a consistent "Tick Tock"—that is, a repeated pattern of one "Tick" followed by one "Tock."

```
// Use Wait() and Pulse() to create a ticking clock.

using System;
using System.Threading;

class TickTock {
  object lockOn = new object();

  public void Tick(bool running) {
    lock(lockOn) {
      if(!running) { // stop the clock
        Monitor.Pulse(lockOn); // notify any waiting threads
        return;
      }

      Console.Write("Tick ");
      Monitor.Pulse(lockOn); // let Tock() run

      Monitor.Wait(lockOn); // wait for Tock() to complete
    }
  }

  public void Tock(bool running) {
    lock(lockOn) {
      if(!running) { // stop the clock
        Monitor.Pulse(lockOn); // notify any waiting threads
        return;
      }

      Console.WriteLine("Tock");
      Monitor.Pulse(lockOn); // let Tick() run

      Monitor.Wait(lockOn); // wait for Tick() to complete
    }
  }
```

```
}

class MyThread {
  public Thread Thrd;
  TickTock ttOb;

  // Construct a new thread.
  public MyThread(string name, TickTock tt) {
    Thrd = new Thread(this.Run);
    ttOb = tt;
    Thrd.Name = name;
    Thrd.Start();
  }

  // Begin execution of new thread.
  void Run() {
    if(Thrd.Name == "Tick") {
      for(int i=0; i<5; i++) ttOb.Tick(true);
      ttOb.Tick(false);
    }
    else {
      for(int i=0; i<5; i++) ttOb.Tock(true);
      ttOb.Tock(false);
    }
  }
}

class TickingClock {
  static void Main() {
    TickTock tt = new TickTock();
    MyThread mt1 = new MyThread("Tick", tt);
    MyThread mt2 = new MyThread("Tock", tt);

    mt1.Thrd.Join();
    mt2.Thrd.Join();
    Console.WriteLine("Clock Stopped");
  }
}
```

Here is the output produced by the program:

```
Tick Tock
Tick Tock
Tick Tock
Tick Tock
Tick Tock
Clock Stopped
```

Let's take a close look at this program. In **Main( )**, a **TickTock** object called **tt** is created, and this object is used to start two threads of execution. Inside the **Run( )** method of **MyThread**, if the name of the thread is "Tick," calls to **Tick( )** are made. If the name of the thread is "Tock," the **Tock( )** method is called. Five calls that pass **true** as an argument are made to each method. The clock runs as long as **true** is passed. A final call that passes **false** to each method stops the clock.

The most important part of the program is found in the **Tick( )** and **Tock( )** methods. We will begin with the **Tick( )** method, which, for convenience, is shown here:

```
public void Tick(bool running) {
  lock(lockOn) {
    if(!running) { // stop the clock
      Monitor.Pulse(lockOn); // notify any waiting threads
      return;
    }

    Console.Write("Tick ");
    Monitor.Pulse(lockOn); // let Tock() run

    Monitor.Wait(lockOn); // wait for Tock() to complete
  }
}
```

First, notice that the code in **Tick( )** is contained within a **lock** block. Recall, **Wait( )** and **Pulse( )** can be used only inside synchronized blocks. The method begins by checking the value of the **running** parameter. This parameter is used to provide a clean shutdown of the clock. If it is **false**, then the clock has been stopped. If this is the case, a call to **Pulse( )** is made to enable any waiting thread to run. We will return to this point in a moment. Assuming the clock is running when **Tick( )** executes, the word "Tick" is displayed, and then a call to **Pulse( )** takes place followed by a call to **Wait( )**. The call to **Pulse( )** allows a thread waiting on the same object to run. The call to **Wait( )** causes **Tick( )** to suspend until another thread calls **Pulse( )**. Thus, when **Tick( )** is called, it displays one "Tick," lets another thread run, and then suspends.

The **Tock( )** method is an exact copy of **Tick( )**, except that it displays "Tock." Thus, when entered, it displays "Tock," calls **Pulse( )**, and then waits. When viewed as a pair, a call to **Tick( )** can be followed only by a call to **Tock( )**, which can be followed only by a call to **Tick( )**, and so on. Therefore, the two methods are mutually synchronized.

The reason for the call to **Pulse( )** when the clock is stopped is to allow a final call to **Wait( )** to succeed. Remember, both **Tick( )** and **Tock( )** execute a call to **Wait( )** after displaying their message. The problem is that when the clock is stopped, one of the methods will still be waiting. Thus, a final call to **Pulse( )** is required in order for the waiting method to run. As an experiment, try removing this call to **Pulse( )** and watch what happens. As you will see, the program will "hang," and you will need to press CTRL-C to exit. The reason for this is that when the final call to **Tock( )** calls **Wait( )**, there is no corresponding call to **Pulse( )** that lets **Tock( )** conclude. Thus, **Tock( )** just sits there, waiting forever.

Before moving on, if you have any doubt that the calls to **Wait( )** and **Pulse( )** are actually needed to make the "clock" run right, substitute this version of **TickTock** into the preceding program. It has all calls to **Wait( )** and **Pulse( )** removed.

```
// A nonfunctional version of TickTock.
class TickTock {

  object lockOn = new object();

  public void Tick(bool running) {
    lock(lockOn) {
      if(!running) { // stop the clock
        return;
```

```
      }

      Console.Write("Tick ");
    }
  }

  public void Tock(bool running) {
    lock(lockOn) {
      if(!running) { // stop the clock
        return;
      }

      Console.WriteLine("Tock");
    }
  }
}
```

After the substitution, the output produced by the program will look like this:

```
Tick Tick Tick Tick Tick Tock
Tock
Tock
Tock
Tock
Clock Stopped
```

Clearly, the **Tick( )** and **Tock( )** methods are no longer synchronized!

Deadlock and Race Conditions

When developing multithreaded programs, you must be careful to avoid deadlock and race conditions. *Deadlock* is, as the name implies, a situation in which one thread is waiting for another thread to do something, but that other thread is waiting on the first. Thus, both threads are suspended, waiting for each other, and neither executes. This situation is analogous to two overly polite people both insisting that the other step through a door first!

Avoiding deadlock seems easy, but it's not. For example, deadlock can occur in roundabout ways. Consider the **TickTock** class. As explained, if a final **Pulse( )** is not executed by **Tick( )** or **Tock( )**, then one or the other will be waiting indefinitely and the program is deadlocked. Often the cause of the deadlock is not readily understood simply by looking at the source code to the program, because concurrently executing threads can interact in complex ways at runtime. To avoid deadlock, careful programming and thorough testing is required. In general, if a multithreaded program occasionally "hangs," deadlock is the likely cause.

A *race condition* occurs when two (or more) threads attempt to access a shared resource at the same time, without proper synchronization. For example, one thread may be writing a new value to a variable while another thread is incrementing the variable's current value. Without synchronization, the outcome will depend on the order in which the threads execute. (Does the second thread increment the original value or the new value written by the first thread?) In situations like this, the two threads are said to be "racing each other," with the final outcome determined by which thread finishes first. Like deadlock, a race condition can occur in difficult-to-discover ways. The solution is prevention: careful programming that properly synchronizes access to shared resources.

Using MethodImplAttribute

It is possible to synchronize an entire method by using the **MethodImplAttribute** attribute. This approach can be used as an alternative to the **lock** statement in cases in which the entire contents of a method are to be locked. **MethodImplAttribute** is defined within the **System.Runtime.CompilerServices** namespace. The constructor that applies to synchronization is shown here:

public MethodImplAttribute(MethodImplOptions *opt*)

Here, *opt* specifies the implementation attribute. To synchronize a method, specify **MethodImplOptions.Synchronized**. This attribute causes the entire method to be locked on the instance (that is, via **this**). (In the case of **static** methods, the type is locked on.) Thus, it must not be used on a public object or with a public class.

Here is a rewrite of the **TickTock** class that uses **MethodImplAttribute** to provide synchronization:

```
// Use MethodImplAttribute to synchronize a method.

using System;
using System.Threading;
using System.Runtime.CompilerServices;

// Rewrite of TickTock to use MethodImplOptions.Synchronized.
class TickTock {

  /* The following attribute synchronizes the entire
     Tick() method. */
  [MethodImplAttribute(MethodImplOptions.Synchronized)]
  public void Tick(bool running) {
    if(!running) { // stop the clock
      Monitor.Pulse(this); // notify any waiting threads
      return;
    }

    Console.Write("Tick ");
    Monitor.Pulse(this); // let Tock() run

    Monitor.Wait(this); // wait for Tock() to complete
  }


  /* The following attribute synchronizes the entire
     Tock() method. */
  [MethodImplAttribute(MethodImplOptions.Synchronized)]
  public void Tock(bool running) {
    if(!running) { // stop the clock
      Monitor.Pulse(this); // notify any waiting threads
      return;
    }

    Console.WriteLine("Tock");
```

```
    Monitor.Pulse(this); // let Tick() run

    Monitor.Wait(this); // wait for Tick() to complete
  }
}

class MyThread {
  public Thread Thrd;
  TickTock ttOb;

  // Construct a new thread.
  public MyThread(string name, TickTock tt) {
    Thrd = new Thread(this.Run);
    ttOb = tt;
    Thrd.Name = name;
    Thrd.Start();
  }

  // Begin execution of new thread.
  void Run() {
    if(Thrd.Name == "Tick") {
      for(int i=0; i<5; i++) ttOb.Tick(true);
      ttOb.Tick(false);
    }
    else {
      for(int i=0; i<5; i++) ttOb.Tock(true);
      ttOb.Tock(false);
    }
  }
}

class TickingClock {
  static void Main() {
    TickTock tt = new TickTock();
    MyThread mt1 = new MyThread("Tick", tt);
    MyThread mt2 = new MyThread("Tock", tt);

    mt1.Thrd.Join();
    mt2.Thrd.Join();
    Console.WriteLine("Clock Stopped");
  }
}
```

The proper Tick Tock output is the same as before.

As long as the method being synchronized is not defined by a public class or called on a public object, then whether you use **lock** or **MethodImplAttribute** is your decision. Both produce the same results. Because **lock** is a keyword built into C#, that is the approach the examples in this book will use.

REMEMBER *Do not use **MethodImplAttribute** with public classes or with public instances. Instead, use **lock**, locking on a private object (as explained earlier).*

Using a Mutex and a Semaphore

Although C#'s **lock** statement is sufficient for many synchronization needs, some situations, such as restricting access to a shared resource, are sometimes more conveniently handled by other synchronization mechanisms built into the .NET Framework. The two described here are related to each other: mutexes and semaphores.

The Mutex

A *mutex* is a mutually exclusive synchronization object. This means it can be acquired by one and only one thread at a time. The mutex is designed for those situations in which a shared resource can be used by only one thread at a time. For example, imagine a log file that is shared by several processes, but only one process can write to that file at any one time. A mutex is the perfect synchronization device to handle this situation.

The mutex is supported by the **System.Threading.Mutex** class. It has several constructors. Two commonly used ones are shown here:

```
public Mutex( )
public Mutex(bool owned)
```

The first version creates a mutex that is initially unowned. In the second version, if *owned* is **true**, the initial state of the mutex is owned by the calling thread. Otherwise, it is unowned.

To acquire the mutex, your code will call **WaitOne( )** on the mutex. This method is inherited by **Mutex** from the **Thread.WaitHandle** class. Here is its simplest form:

```
public bool WaitOne( );
```

It waits until the mutex on which it is called can be acquired. Thus, it blocks execution of the calling thread until the specified mutex is available. It always returns true.

When your code no longer needs ownership of the mutex, it releases it by calling **ReleaseMutex( )**, shown here:

```
public void ReleaseMutex( )
```

This releases the mutex on which it is called, enabling the mutex to be acquired by another thread.

To use a mutex to synchronize access to a shared resource, you will use **WaitOne( )** and **ReleaseMutex( )**, as shown in the following sequence:

```
Mutex myMtx = new Mutex();

// ...

myMtx.WaitOne(); // wait to acquire the mutex

// Access the shared resource.

myMtx.ReleaseMutex(); // release the mutex
```

When the call to **WaitOne( )** takes place, execution of the thread will suspend until the mutex can be acquired. When the call to **ReleaseMutex( )** takes place, the mutex is released and another thread can acquire it. Using this approach, access to a shared resource can be limited to one thread at a time.

The following program puts this framework into action. It creates two threads, **IncThread** and **DecThread**, which both access a shared resource called **SharedRes.Count**. **IncThread** increments **SharedRes.Count** and **DecThread** decrements it. To prevent both threads from accessing **SharedRes.Count** at the same time, access is synchronized by the **Mtx** mutex, which is also part of the **SharedRes** class.

```
// Use a Mutex.

using System;
using System.Threading;

// This class contains a shared resource (Count),
// and a mutex (Mtx) to control access to it.
class SharedRes {
  public static int Count = 0;
  public static Mutex Mtx = new Mutex();
}

// This thread increments SharedRes.Count.
class IncThread {
  int num;
  public Thread Thrd;

  public IncThread(string name, int n) {
    Thrd = new Thread(this.Run);
    num = n;
    Thrd.Name = name;
    Thrd.Start();
  }

  // Entry point of thread.
  void Run() {

    Console.WriteLine(Thrd.Name + " is waiting for the mutex.");

    // Acquire the Mutex.
    SharedRes.Mtx.WaitOne();

    Console.WriteLine(Thrd.Name + " acquires the mutex.");

    do {
      Thread.Sleep(500);
      SharedRes.Count++;
      Console.WriteLine("In " + Thrd.Name +
                        ", SharedRes.Count is " + SharedRes.Count);
      num--;
    } while(num > 0);

    Console.WriteLine(Thrd.Name + " releases the mutex.");

    // Release the Mutex.
    SharedRes.Mtx.ReleaseMutex();
  }
}
```

```
// This thread decrements SharedRes.Count.
class DecThread {
  int num;
  public Thread Thrd;

  public DecThread(string name, int n) {
    Thrd = new Thread(new ThreadStart(this.Run));
    num = n;
    Thrd.Name = name;
    Thrd.Start();
  }

  // Entry point of thread.
  void Run() {

    Console.WriteLine(Thrd.Name + " is waiting for the mutex.");

    // Acquire the Mutex.
    SharedRes.Mtx.WaitOne();

    Console.WriteLine(Thrd.Name + " acquires the mutex.");

    do {
      Thread.Sleep(500);
      SharedRes.Count--;
      Console.WriteLine("In " + Thrd.Name +
                        ", SharedRes.Count is " + SharedRes.Count);
      num--;
    } while(num > 0);

    Console.WriteLine(Thrd.Name + " releases the mutex.");

    // Release the Mutex.
    SharedRes.Mtx.ReleaseMutex();
  }
}

class MutexDemo {
  static void Main() {

    // Construct three threads.
    IncThread mt1 = new IncThread("Increment Thread", 5);

    Thread.Sleep(1); // let the Increment thread start

    DecThread mt2 = new DecThread("Decrement Thread", 5);

    mt1.Thrd.Join();
    mt2.Thrd.Join();
  }
}
```

The output is shown here:

```
Increment Thread is waiting for the mutex.
Increment Thread acquires the mutex.
```

```
Decrement Thread is waiting for the mutex.
In Increment Thread, SharedRes.Count is 1
In Increment Thread, SharedRes.Count is 2
In Increment Thread, SharedRes.Count is 3
In Increment Thread, SharedRes.Count is 4
In Increment Thread, SharedRes.Count is 5
Increment Thread releases the mutex.
Decrement Thread acquires the mutex.
In Decrement Thread, SharedRes.Count is 4
In Decrement Thread, SharedRes.Count is 3
In Decrement Thread, SharedRes.Count is 2
In Decrement Thread, SharedRes.Count is 1
In Decrement Thread, SharedRes.Count is 0
Decrement Thread releases the mutex.
```


As the output shows, access to **SharedRes.Count** is synchronized, with only one thread at a time being able to change its value.

To prove that the **Mtx** mutex was needed to produce the preceding output, try commenting out the calls to **WaitOne( )** and **ReleaseMutex( )** in the preceding program. When you run the program, you will see the following sequence (the actual output you see may vary):

```
In Increment Thread, SharedRes.Count is 1
In Decrement Thread, SharedRes.Count is 0
In Increment Thread, SharedRes.Count is 1
In Decrement Thread, SharedRes.Count is 0
In Increment Thread, SharedRes.Count is 1
In Decrement Thread, SharedRes.Count is 0
In Increment Thread, SharedRes.Count is 1
In Decrement Thread, SharedRes.Count is 0
In Increment Thread, SharedRes.Count is 1
```

As this output shows, without the mutex, increments and decrements to **SharedRes.Count** are interspersed rather than sequenced.

The mutex created by the previous example is known only to the process that creates it. However, it is possible to create a mutex that is known systemwide. To do so, you must create a named mutex, using one of these constructors:

public Mutex(bool *owned*, string *name*)
public Mutex(bool *owned*, string *name*, out bool *whatHappened*)

In both forms, the name of the mutex is passed in *name*. In the first form, if *owned* is **true**, then ownership of the mutex is requested. However, because a systemwide mutex might already be owned by another process, it is better to specify **false** for this parameter. In the second form, on return, *whatHappened* will be **true** if ownership was requested and acquired. It will be **false** if ownership was denied. (There is also a third form of the **Mutex** constructor that allows you to specify a **MutexSecurity** object, which controls access.) Using a named mutex enables you to manage interprocess synchronization.

One other point: It is legal for a thread that has acquired a mutex to make one or more additional calls to **WaitOne( )** prior to calling **ReleaseMutex( )**, and these additional calls will succeed. That is, redundant calls to **WaitOne( )** will not block a thread that already owns the mutex. However, the number of calls to **WaitOne( )** must be balanced by the same number of calls to **ReleaseMutex( )** before the mutex is released.

The Semaphore

A semaphore is similar to a mutex except that it can grant more than one thread access to a shared resource at the same time. Thus, the semaphore is useful when a collection of resources is being synchronized. A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, access is denied. What the counter is counting are *permits.* Thus, to access the resource, a thread must be granted a permit from the semaphore.

In general, to use a semaphore, the thread that wants access to the shared resource tries to acquire a permit. If the semaphore's counter is greater than zero, the thread acquires a permit, which causes the semaphore's count to be decremented. Otherwise, the thread will block until a permit can be acquired. When the thread no longer needs access to the shared resource, it releases the permit, which causes the semaphore's count to be incremented. If there is another thread waiting for a permit, then that thread will acquire a permit at that time. The number of simultaneous accesses permitted is specified when the semaphore is created. If you create a semaphore that allows only one access, then a semaphore acts just like a mutex.

Semaphores are especially useful in situations in which a shared resource consists of a group or *pool.* For example, a collection of network connections, any of which can be used for communication, is a resource pool. A thread needing a network connection doesn't care which one it gets. In this case, a semaphore offers a convenient mechanism to manage access to the connections.

The semaphore is implemented by **System.Threading.Semaphore**. It has several constructors. The simplest form is shown here:

public Semaphore(int *initial*, int *max*)

Here, *initial* specifies the initial value of the semaphore permit counter, which is the number of permits available. The maximum value of the counter is passed in *max*. Thus, *max* represents the maximum number of permits that can granted by the semaphore. The value in *initial* specifies how many of these permits are initially available.

Using a semaphore is similar to using a mutex, described earlier. To acquire access, your code will call **WaitOne( )** on the semaphore. This method is inherited by **Semaphore** from the **WaitHandle** class. **WaitOne( )** waits until the semaphore on which it is called can be acquired. Thus, it blocks execution of the calling thread until the specified semaphore can grant permission.

When your code no longer needs ownership of the semaphore, it releases it by calling **Release( )**, which is shown here:

public int Release( )
public int Release(int *num*)

The first form releases one permit. The second form releases the number of permits specified by *num.* Both return the permit count that existed prior to the release.

It is possible for a thread to call **WaitOne( )** more than once before calling **Release( )**. However, the number of calls to **WaitOne( )** must be balanced by the same number of calls to **Release( )** before the permit is released. Alternatively, you can call the **Release(int)** form, passing a number equal to the number of times that **WaitOne( )** was called.

Here is an example that illustrates the semaphore. In the program, the class **MyThread** uses a semaphore to allow only two **MyThread** threads to be executed at any one time. Thus, the resource being shared is the CPU.

```
// Use a Semaphore.

using System;
using System.Threading;

// This thread allows only two instances of itself
// to run at any one time.
class MyThread {
  public Thread Thrd;

  // This creates a semaphore that allows up to two
  // permits to be granted and that initially has
  // two permits available.
  static Semaphore sem = new Semaphore(2, 2);

  public MyThread(string name) {
    Thrd = new Thread(this.Run);
    Thrd.Name = name;
    Thrd.Start();
  }

  // Entry point of thread.
  void Run() {

    Console.WriteLine(Thrd.Name + " is waiting for a permit.");

    sem.WaitOne();

    Console.WriteLine(Thrd.Name + " acquires a permit.");

    for(char ch='A'; ch < 'D'; ch++) {
      Console.WriteLine(Thrd.Name + " : " + ch + " ");
      Thread.Sleep(500);
    }

    Console.WriteLine(Thrd.Name + " releases a permit.");

    // Release the semaphore.
    sem.Release();
  }
}


class SemaphoreDemo {
  static void Main() {

    // Construct three threads.
```

```
    MyThread mt1 = new MyThread("Thread #1");
    MyThread mt2 = new MyThread("Thread #2");
    MyThread mt3 = new MyThread("Thread #3");

    mt1.Thrd.Join();
    mt2.Thrd.Join();
    mt3.Thrd.Join();
  }
}
```

MyThread declares the semaphore **sem**, as shown here:

```
static Semaphore sem = new Semaphore(2, 2);
```

This creates a semaphore that can grant up to two permits and that initially has both permits available.

In **MyThread.Run( )**, notice that execution cannot continue until a permit is granted by the semaphore, **sem**. If no permits are available, then execution of that thread suspends. When a permit does become available, execution resumes and the thread can run. In **Main( )**, three **MyThread** threads are created. However, only the first two get to execute. The third must wait until one of the other threads terminates. The output, shown here, verifies this. (The actual output you see may vary slightly.)

```
Thread #1 is waiting for a permit.
Thread #1 acquires a permit.
Thread #1 : A
Thread #2 is waiting for a permit.
Thread #2 acquires a permit.
Thread #2 : A
Thread #3 is waiting for a permit.
Thread #1 : B
Thread #2 : B
Thread #1 : C
Thread #2 : C
Thread #1 releases a permit.
Thread #3 acquires a permit.
Thread #3 : A
Thread #2 releases a permit.
Thread #3 : B
Thread #3 : C
Thread #3 releases a permit.
```

The semaphore created by the previous example is known only to the process that creates it. However, it is possible to create a semaphore that is known systemwide. To do so, you must create a named semaphore. To do this, use one of these constructors:

public Semaphore(int *initial*, int *max*, string *name*)
public Semaphore(int *initial*, int *max*, string *name*, out bool *whatHappened*)

In both forms, the name of the semaphore is passed in *name.* In the first form, if a semaphore by the specified name does not already exist, it is created using the values of *initial* and *max.* If it does already exist, then the values of *initial* and *max* are ignored. In the second form, on return, *whatHappened* will be **true** if the semaphore was created. In this case, the values of *initial* and *max* will be used to create the semaphore. If *whatHappened* is **false**, then the

semaphore already exists and the values of *initial* and *max* are ignored. (There is also a third form of the **Semaphore** constructor that allows you to specify a **SemaphoreSecurity** object, which controls access.) Using a named semaphore enables you to manage interprocess synchronization.

Using Events

C# supports another type of synchronization object: the event. There are two types of events: manual reset and auto reset. These are supported by the classes **ManualResetEvent** and **AutoResetEvent**. These classes are derived from the top-level class **EventWaitHandle**. These classes are used in situations in which one thread is waiting for some event to occur in another thread. When the event takes place, the second thread signals the first, allowing it to resume execution.

The constructors for **ManualResetEvent** and **AutoResetEvent** are shown here:

public ManualResetEvent(bool *status*)
public AutoResetEvent(bool *status*)

Here, if *status* is **true**, the event is initially signaled. If *status* is **false**, the event is initially non-signaled.

Events are easy to use. For a **ManualResetEvent**, the procedure works like this. A thread that is waiting for some event simply calls **WaitOne( )** on the event object representing that event. **WaitOne( )** returns immediately if the event object is in a signaled state. Otherwise, it suspends execution of the calling thread until the event is signaled. After another thread performs the event, that thread sets the event object to a signaled state by calling **Set( )**. Thus, a call to **Set( )** can be understood as signaling that an event has occurred. After the event object is set to a signaled state, the call to **WaitOne( )** will return and the first thread will resume execution. The event is returned to a non-signaled state by calling **Reset( )**.

The difference between **AutoResetEvent** and **ManualResetEvent** is how the event gets reset. For **ManualResetEvent**, the event remains signaled until a call to **Reset( )** is made. For **AutoResetEvent**, the event automatically changes to a non-signaled state as soon as a thread waiting on that event receives the event notification and resumes execution. Thus, a call to **Reset( )** is not necessary when using **AutoResetEvent**.

Here is an example that illustrates **ManualResetEvent**:

```
// Use a manual event object.

using System;
using System.Threading;

// This thread signals the event passed to its constructor.
class MyThread {
  public Thread Thrd;
  ManualResetEvent mre;

  public MyThread(string name, ManualResetEvent evt) {
    Thrd = new Thread(this.Run);
    Thrd.Name = name;
    mre = evt;
    Thrd.Start();
  }
```

```
  // Entry point of thread.
  void Run() {
    Console.WriteLine("Inside thread " + Thrd.Name);

    for(int i=0; i<5; i++) {
      Console.WriteLine(Thrd.Name);
      Thread.Sleep(500);
    }

    Console.WriteLine(Thrd.Name + " Done!");

    // Signal the event.
    mre.Set();
  }
}

class ManualEventDemo {
  static void Main() {
    ManualResetEvent evtObj = new ManualResetEvent(false);

    MyThread mt1 = new MyThread("Event Thread 1", evtObj);

    Console.WriteLine("Main thread waiting for event.");

    // Wait for signaled event.
    evtObj.WaitOne();

    Console.WriteLine("Main thread received first event.");


    // Reset the event.
    evtObj.Reset();


    mt1 = new MyThread("Event Thread 2", evtObj);

    // Wait for signaled event.
    evtObj.WaitOne();

    Console.WriteLine("Main thread received second event.");
  }
}
```

The output is shown here. (The actual output you see may vary slightly.)

```
Inside thread Event Thread 1
Event Thread 1
Main thread waiting for event.
Event Thread 1
Event Thread 1
Event Thread 1
Event Thread 1
Event Thread 1 Done!
Main thread received first event.
Inside thread Event Thread 2
```

```
Event Thread 2
Event Thread 2
Event Thread 2
Event Thread 2
Event Thread 2
Event Thread 2 Done!
Main thread received second event.
```

First, notice that **MyThread** is passed a **ManualResetEvent** in its constructor. When **MyThread**'s **Run( )** method finishes, it calls **Set( )** on that event object, which puts the event object into a signaled state. Inside **Main( )**, a **ManualResetEvent** called **evtObj** is created with an initially unsignaled state. Then, a **MyThread** instance is created and passed **evtObj**. Next, the main thread waits on the event object. Because the initial state of **evtObj** is not signaled, this causes the main thread to wait until the instance of **MyThread** calls **Set( )**, which puts **evtObj** into a signaled state. This allows the main thread to run again. Then the event is reset and the process is repeated for the second thread. Without the use of the event object, all threads would have run simultaneously and their output would have been jumbled. To verify this, try commenting out the call to **WaitOne( )** inside **Main( )**.

In the preceding program, if an **AutoResetEvent** object rather than a **ManualResetEvent** object were used, then the call to **Reset( )** in **Main( )** would not be necessary. The reason is that the event is automatically set to a non-signaled state when a thread waiting on the event is resumed. To try this, simply change all references to **ManualResetEvent** to **AutoResetEvent** and remove the calls to **Reset( )**. This version will execute the same as before.

The Interlocked Class

One other class that is related to synchronization is **Interlocked**. This class offers an alternative to the other synchronization features when all you need to do is change the value of a shared variable. The methods provided by **Interlocked** guarantee that their operation is performed as a single, uninterruptable operation. Thus, no other synchronization is needed. **Interlocked** provides static methods that add two integers, increment an integer, decrement an integer, compare and set an object, exchange objects, and obtain a 64-bit value. All of these operations take place without interruption.

The following program demonstrates two **Interlocked** methods: **Increment( )** and **Decrement( )**. Here the forms of these methods that will be used:

public static int Increment(ref int *v*)

public static int Decrement(ref int *v*)

Here, *v* is the value to be incremented or decremented.

```
// Use Interlocked operations.

using System;
using System.Threading;

// A shared resource.
class SharedRes {
  public static int Count = 0;
}
```

```
// This thread increments SharedRes.Count.
class IncThread {
  public Thread Thrd;

  public IncThread(string name) {
    Thrd = new Thread(this.Run);
    Thrd.Name = name;
    Thrd.Start();
  }

  // Entry point of thread.
  void Run() {

    for(int i=0; i<5; i++) {
      Interlocked.Increment(ref SharedRes.Count);
      Console.WriteLine(Thrd.Name + " Count is " + SharedRes.Count);
    }
  }
}

// This thread decrements SharedRes.Count.
class DecThread {
  public Thread Thrd;

  public DecThread(string name) {
    Thrd = new Thread(this.Run);
    Thrd.Name = name;
    Thrd.Start();
  }

  // Entry point of thread.
  void Run() {

    for(int i=0; i<5; i++) {
      Interlocked.Decrement(ref SharedRes.Count);
      Console.WriteLine(Thrd.Name + " Count is " + SharedRes.Count);
    }
  }
}

class InterlockedDemo {
  static void Main() {

    // Construct two threads.
    IncThread mt1 = new IncThread("Increment Thread");
    DecThread mt2 = new DecThread("Decrement Thread");

    mt1.Thrd.Join();
    mt2.Thrd.Join();
  }
}
```

Terminating a Thread

It is sometimes useful to stop a thread prior to its normal conclusion. For example, a debugger may need to stop a thread that has run wild. Once a thread has been terminated, it is removed from the system and cannot be restarted.

To terminate a thread prior to its normal ending point, use **Thread.Abort( )**. Its simplest form is shown here:

```
public void Abort( )
```

Abort( ) causes a **ThreadAbortException** to be thrown to the thread on which **Abort( )** is called. This exception causes the thread to terminate. This exception can also be caught by your code (but is automatically rethrown in order to stop the thread). **Abort( )** may not always be able to stop a thread immediately, so if it is important that a thread be stopped before your program continues, you will need to follow a call to **Abort( )** with a call to **Join( )**. Also, in rare cases, it is possible that **Abort( )** won't be able to stop a thread. One way this could happen is if a **finally** block goes into an infinite loop.

The following example shows how to stop a thread by use of **Abort( )**:

```
// Stopping a thread by use of Abort().

using System;
using System.Threading;

class MyThread {
  public Thread Thrd;

  public MyThread(string name) {
    Thrd = new Thread(this.Run);
    Thrd.Name = name;
    Thrd.Start();
  }

  // This is the entry point for thread.
  void Run() {
    Console.WriteLine(Thrd.Name + " starting.");

    for(int i = 1; i <= 1000; i++) {
      Console.Write(i + " ");
      if((i%10)==0) {
        Console.WriteLine();
        Thread.Sleep(250);
      }
    }
    Console.WriteLine(Thrd.Name + " exiting.");
  }
}

class StopDemo {
  static void Main() {
```

```
    MyThread mt1 = new MyThread("My Thread");

    Thread.Sleep(1000); // let child thread start executing

    Console.WriteLine("Stopping thread.");
    mt1.Thrd.Abort();

    mt1.Thrd.Join(); // wait for thread to terminate

    Console.WriteLine("Main thread terminating.");
  }
}
```

The output from this program is shown here:

```
My Thread starting.
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
Stopping thread.
Main thread terminating.
```

NOTE *Abort( ) should not be used as the normal means of stopping a thread. It is meant for specialized situations. Usually, a thread should end because its entry point method returns.*

An Abort( ) Alternative

You might find a second form of **Abort( )** useful in some cases. Its general form is shown here:

public void Abort(object *info*)

Here, *info* contains any information that you want to pass to the thread when it is being stopped. This information is accessible through the **ExceptionState** property of **ThreadAbortException**. You might use this to pass a termination code to a thread. The following program demonstrates this form of **Abort( )**:

```
// Using Abort(object).

using System;
using System.Threading;

class MyThread {
  public Thread Thrd;

  public MyThread(string name) {
    Thrd = new Thread(this.Run);
    Thrd.Name = name;
    Thrd.Start();
  }

  // This is the entry point for thread.
  void Run() {
    try {
```

```
      Console.WriteLine(Thrd.Name + " starting.");

      for(int i = 1; i <= 1000; i++) {
        Console.Write(i + " ");
        if((i%10)==0) {
          Console.WriteLine();
          Thread.Sleep(250);
        }
      }
      Console.WriteLine(Thrd.Name + " exiting normally.");
    } catch(ThreadAbortException exc) {
      Console.WriteLine("Thread aborting, code is " +
                        exc.ExceptionState);
    }
  }
}

class UseAltAbort {
  static void Main() {
    MyThread mt1 = new MyThread("My Thread");

    Thread.Sleep(1000); // let child thread start executing

    Console.WriteLine("Stopping thread.");
    mt1.Thrd.Abort(100);

    mt1.Thrd.Join(); // wait for thread to terminate

    Console.WriteLine("Main thread terminating.");
  }
}
```

The output is shown here:

```
My Thread starting.
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
Stopping thread.
Thread aborting, code is 100
Main thread terminating.
```

As the output shows, the value 100 is passed to **Abort( )**. This value is then accessed through the **ExceptionState** property of the **ThreadAbortException** caught by the thread when it is terminated.

Canceling Abort( )

A thread can override a request to abort. To do so, the thread must catch the **ThreadAbortException** and then call **ResetAbort( )**. This prevents the exception from being automatically rethrown when the thread's exception handler ends. **ResetAbort( )** is declared like this:

```
public static void ResetAbort( )
```

A call to **ResetAbort( )** can fail if the thread does not have the proper security setting to cancel the abort.

The following program demonstrates **ResetAbort( )**:

```
// Using ResetAbort().

using System;
using System.Threading;

class MyThread {
  public Thread Thrd;

  public MyThread(string name) {
    Thrd = new Thread(this.Run);
    Thrd.Name = name;
    Thrd.Start();
  }

  // This is the entry point for thread.
  void Run() {
    Console.WriteLine(Thrd.Name + " starting.");

    for(int i = 1; i <= 1000; i++) {
      try {
        Console.Write(i + " ");
        if((i%10)==0) {
          Console.WriteLine();
          Thread.Sleep(250);
        }
      } catch(ThreadAbortException exc) {
        if((int)exc.ExceptionState == 0) {
          Console.WriteLine("Abort Cancelled! Code is " +
                            exc.ExceptionState);
          Thread.ResetAbort();
        }
        else
          Console.WriteLine("Thread aborting, code is " +
                            exc.ExceptionState);
      }
    }
    Console.WriteLine(Thrd.Name + " exiting normally.");
  }
}

class ResetAbort {
  static void Main() {
    MyThread mt1 = new MyThread("My Thread");

    Thread.Sleep(1000); // let child thread start executing

    Console.WriteLine("Stopping thread.");
    mt1.Thrd.Abort(0); // this won't stop the thread
```

```
    Thread.Sleep(1000); // let child execute a bit longer

    Console.WriteLine("Stopping thread.");
    mt1.Thrd.Abort(100); // this will stop the thread

    mt1.Thrd.Join(); // wait for thread to terminate

    Console.WriteLine("Main thread terminating.");
  }
}
```

The output is shown here:

```
My Thread starting.
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
Stopping thread.
Abort Cancelled! Code is 0
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
Stopping thread.
Thread aborting, code is 100
Main thread terminating.
```

In this example, if **Abort( )** is called with an argument that equals zero, then the abort request is cancelled by the thread by calling **ResetAbort( )**, and the thread's execution continues. Any other value causes the thread to stop.

Suspending and Resuming a Thread

In early versions of the .NET Framework, a thread could be suspended by calling **Thread.Suspend( )** and resumed by calling **Thread.Resume( )**. Today, however, both of these methods are marked as obsolete and should not be used for new code. One reason is that **Suspend( )** is inherently dangerous because it can be used to suspend a thread that is currently holding a lock, thus preventing the lock from being released, resulting in deadlock. This can cause a systemwide problem. You must use C#'s other synchronization features, such as a mutex, to suspend and resume a thread.

Determining a Thread's State

The state of a thread can be obtained from the **ThreadState** property provided by **Thread**. It is shown here:

```
public ThreadState ThreadState{ get; }
```

The state of the thread is returned as a value defined by the **ThreadState** enumeration. It defines the following values:

ThreadState.Aborted	ThreadState.AbortRequested
ThreadState.Background	ThreadState.Running
ThreadState.Stopped	ThreadState.StopRequested
ThreadState.Suspended	ThreadState.SuspendRequested
ThreadState.Unstarted	ThreadState.WaitSleepJoin

All but one of these values is self-explanatory. The one that needs some explanation is **ThreadState.WaitSleepJoin**. A thread enters this state when it is waiting because of a call to **Wait( )**, **Sleep( )**, or **Join( )**.

Using the Main Thread

As mentioned at the start of this chapter, all C# programs have at least one thread of execution, called the *main thread*, which is given to the program automatically when it begins running. The main thread can be handled just like all other threads.

To access the main thread, you must obtain a **Thread** object that refers to it. You do this through the **CurrentThread** property, which is a member of **Thread**. Its general form is shown here:

```
public static Thread CurrentThread{ get; }
```

This property returns a reference to the thread in which it is used. Therefore, if you use **CurrentThread** while execution is inside the main thread, you will obtain a reference to the main thread. Once you have this reference, you can control the main thread just like any other thread.

The following program obtains a reference to the main thread and then gets and sets the main thread's name and priority:

```
// Control the main thread.

using System;
using System.Threading;

class UseMain {
  static void Main() {
    Thread Thrd;

    // Get the main thread.
    Thrd = Thread.CurrentThread;

    // Display main thread's name.
    if(Thrd.Name == null)
      Console.WriteLine("Main thread has no name.");
    else
      Console.WriteLine("Main thread is called: " + Thrd.Name);

    // Display main thread's priority.
    Console.WriteLine("Priority: " + Thrd.Priority);
```

```
    Console.WriteLine();

    // Set the name and priority.
    Console.WriteLine("Setting name and priority.\n");
    Thrd.Name = "Main Thread";
    Thrd.Priority = ThreadPriority.AboveNormal;

    Console.WriteLine("Main thread is now called: " +
                      Thrd.Name);

    Console.WriteLine("Priority is now: " +
                      Thrd.Priority);
  }
}
```

The output from the program is shown here:

```
Main thread has no name.
Priority: Normal

Setting name and priority.

Main thread is now called: Main Thread
Priority is now: AboveNormal
```

One word of caution: You need to be careful about what operations you perform on the main thread. For example, if you add this call to **Join( )** to the end of **Main( )**,

```
Thrd.Join();
```

the program will never terminate because it will be waiting for the main thread to end!

Multithreading Tips

The key to effectively utilizing multithreading is to think concurrently rather than serially. For example, when you have two subsystems within a program that can execute concurrently, make them into individual threads. A word of caution is in order, however. If you create too many threads, you can actually degrade your program's performance rather than enhance it. Remember, there is some overhead associated with context switching. If you create too many threads, more CPU time will be spent changing contexts than in executing your program!

Starting a Separate Task

Although thread-based multitasking is what you will use most often when programming in C#, it is possible to utilize process-based multitasking where appropriate. When using process-based multitasking, instead of starting another thread within the same program, one program starts the execution of another program. In C#, you do this by using the **Process** class. **Process** is defined within the **System.Diagnostics** namespace. To conclude this chapter, a brief look at starting and managing another process is offered.

The easiest way to start another process is to use the **Start( )** method defined by **Process**. Here is one of its simplest forms:

public static Process Start(string *name*)

Here, *name* specifies the name of an executable file that will be executed or a file that is associated with an executable.

When a process that you create ends, call **Close( )** to free the memory associated with that process. It is shown here:

public void Close( )

You can terminate a process in two ways. If the process is a Windows GUI application, then to terminate the process, call **CloseMainWindow( )**, shown here:

public bool CloseMainWindow( )

This method sends a message to the process, instructing it to stop. It returns true if the message was received. It returns false if the application was not a GUI application, or does not have a main window. Furthermore, **CloseMainWindow( )** is only a request to shut down. If the application ignores the request, the application will not be terminated.

To positively terminate a process, call **Kill( )**, as shown here:

public void Kill( )

Use **Kill( )** carefully. It causes an uncontrolled termination of the process. Any unsaved data associated with the process will most likely be lost.

You can wait for a process to end by calling **WaitForExit( )**. Its two forms are shown here:

public void WaitForExit( )
public bool WaitForExit(int *milliseconds*)

The first form waits until the process terminates. The second waits for only the specified number of milliseconds. The second form returns true if the process has terminated and false if it is still running.

The following program demonstrates how to create, wait for, and close a process. It starts the standard Windows utility program **WordPad.exe**. It then waits for WordPad to end.

```
// Starting a new process.

using System;
using System.Diagnostics;

class StartProcess {
  static void Main() {
    Process newProc = Process.Start("wordpad.exe");

    Console.WriteLine("New process started.");

    newProc.WaitForExit();

    newProc.Close(); // free resources

    Console.WriteLine("New process ended.");
  }
}
```

When you run this program, WordPad will start, and you will see the message, "New process started." The program will then wait until you close WordPad. Once WordPad has been terminated, the final message "New process ended." is displayed.

24
CHAPTER

Collections, Enumerators, and Iterators

This chapter discusses one of the most important parts of the .NET Framework: collections. In C#, a *collection* is a group of objects. The .NET Framework contains a large number of interfaces and classes that define and implement various types of collections. Collections simplify many programming tasks because they provide off-the-shelf solutions to several common, but sometimes tedious-to-develop, data structures. For example, there are built-in collections that support dynamic arrays, linked lists, stacks, queues, and hash tables. Collections are a state-of-the-art technology that merits close attention by all C# programmers.

Originally, there were only non-generic collection classes. However, the addition of generics in C# 2.0 coincided with the addition of many new generic classes and interfaces to the .NET Framework. The inclusion of the generic collections essentially doubled the number of collection classes and interfaces. Thus, the Collections API is now quite large. Although the generic and non-generic collections work in similar ways, there are some differences, and both are described in this chapter.

Also described in this chapter are two features that relate to collections: enumerators and iterators. Both enumerators and iterators enable the contents of a class to be cycled through via a **foreach** loop.

Collections Overview

The principal benefit of collections is that they standardize the way groups of objects are handled by your programs. All collections are designed around a set of cleanly defined interfaces. Several built-in implementations of these interfaces, such as **ArrayList**, **Hashtable**, **Stack**, and **Queue**, are provided, which you can use as-is. You can also implement your own collection, but you will seldom need to.

The .NET Framework supports four general types of collections: non-generic, specialized, bit based, and generic. The non-generic collections implement several fundamental data structures, including a dynamic array, stack, and queue. They also include *dictionaries,* in which you can store key/value pairs. An essential point to understand about the non-generic collections is that they operate on data of type **object**.

Thus, they can be used to store any type of data, and different types of data can be mixed within the same collection. Of course, because they store **object** references, they are not type-safe. The non-generic collection classes and interfaces are in **System.Collections**.

The specialized collections operate on a specific type of data or operate in a unique way. For example, there are specialized collections for strings. There are also specialized collections that use a singly linked list. The specialized collections are declared in **System.Collections.Specialized**.

The Collections API defines one bit-based collection called **BitArray**. **BitArray** supports bitwise operations on bits, such as AND and XOR. As such, it differs significantly in its capabilities from the other collections. **BitArray** is declared in **System.Collections**.

The generic collections provide generic implementations of several standard data structures, such as linked lists, stacks, queues, and dictionaries. Because these collections are generic, they are type-safe. This means that only items that are type-compatible with the type of the collection can be stored in a generic collection, thus eliminating accidental type mismatches. Generic collections are declared in **System.Collections.Generic**.

There are also several classes in the **System.Collections.ObjectModel** namespace that support programmers who want to create their own generic collections.

Fundamental to all collections is the concept of an *enumerator,* which is supported by the non-generic interfaces **IEnumerator** and **IEnumerable**, and the generic interfaces **IEnumerator<T>** and **IEnumerable<T>**. An enumerator provides a standardized way of accessing the elements within a collection, one at a time. Thus, it *enumerates* the contents of a collection. Because each collection must implement either a generic or non-generic form of **IEnumerable**, the elements of any collection class can be accessed through the methods defined by **IEnumerator** or **IEnumerator<T>**. Therefore, with only small changes, the code that cycles through one type of collection can be used to cycle through another. As a point of interest, the **foreach** loop uses the enumerator to cycle through the contents of a collection.

A feature related to an enumerator is the *iterator.* It simplifies the process of creating classes, such as custom collections, that can be cycled through by a **foreach** loop. Iterators are also described in this chapter.

One last thing: If you are familiar with C++, then you will find it helpful to know that the collection classes are similar in spirit to the Standard Template Library (STL) classes defined by C++. What a C++ programmer calls a *container,* a C# programmer calls a *collection*. The same is true of Java. If you are familiar with Java's Collections Framework, then you will have no trouble learning to use C# collections.

Because of the differences among the four types of collections—non-generic, bit-based, specialized, and generic—this chapter discusses each separately.

The Non-Generic Collections

The non-generic collections have been part of the .NET Framework since version 1.0. They are defined in the **System.Collections** namespace. The non-generic collections are general-purpose data structures that operate on **object** references. Thus, they can manage any type of object, but not in a type-safe manner. This is both their advantage and disadvantage. Because they operate on **object** references, you can mix various types of data within the same collection. This makes them useful in situations in which you need to manage a collection of different types of objects or when the type of objects being stored are not known in advance. However, if you intend a collection to store a specific type of object, then the non-generic collections do not have the type safety that is found in the generic collections.

The non-generic collections are defined by a set of interfaces and the classes that implement those interfaces. Each is described by the following sections.

The Non-Generic Interfaces

System.Collections defines a number of non-generic interfaces. It is necessary to begin with the collection interfaces because they determine the functionality common to all of the non-generic collection classes. The interfaces that underpin collections are summarized in Table 24-1. The following sections examine each interface in detail.

The ICollection Interface

The **ICollection** interface is the foundation upon which all non-generic collections are built. It declares the core methods and properties that all non-generic collections will have. It also inherits the **IEnumerable** interface.

ICollection defines the following properties:

Property	Meaning
int Count { get; }	The number of items currently held in the collection.
bool IsSynchronized { get; }	Is true if the collection is synchronized and false if it is not. By default, collections are not synchronized. It is possible, though, to obtain a synchronized version of most collections.
object SyncRoot { get; }	An object upon which the collection can be synchronized.

Count is the most often used property because it contains the number of elements currently held in a collection. If **Count** is zero, then the collection is empty.

ICollection defines the following method:

void CopyTo(Array *target*, int *startIdx*)

CopyTo( ) copies the contents of a collection to the array specified by *target,* beginning at the index specified by *startIdx.* Thus, **CopyTo( )** provides a pathway from a collection to a standard C# array.

Interface	Description
ICollection	Defines the elements that all non-generic collections must have.
IComparer	Defines the **Compare( )** method that performs a comparison on objects stored in a collection.
IDictionary	Defines a collection that consists of key/value pairs.
IDictionaryEnumerator	Defines the enumerator for a collection that implements **IDictionary**.
IEnumerable	Defines the **GetEnumerator( )** method, which supplies the enumerator for a collection class.
IEnumerator	Provides methods that enable the contents of a collection to be obtained one at a time.
IEqualityComparer	Compares two objects for equality.
IHashCodeProvider	Declared obsolete. Use **IEqualityComparer** instead.
IList	Defines a collection that can be accessed via an indexer.

Table 24-1 The Non-Generic Collection Interfaces

Because **ICollection** inherits **IEnumerable**, it also includes the sole method defined by **IEnumerable**: **GetEnumerator( )**, which is shown here:

IEnumerator GetEnumerator( )

It returns the enumerator for the collection.

Because **ICollection** inherits **IEnumerable**, three extension methods are defined for it. They are **AsQueryable( )**, **Cast( )**, and **OfType( )**. **AsQueryable( )** is declared in **System.Linq.Queryable**. Both **Cast( )** and **OfType( )** are declared in **System.Linq.Enumerable**. These methods are designed primarily to support LINQ, but may be useful in other contexts.

The IList Interface

The **IList** interface declares the behavior of a non-generic collection that allows elements to be accessed via a zero-based index. It inherits **ICollection** and **IEnumerable**. In addition to the methods defined by **ICollection** and **IEnumerable**, **IList** defines several of its own. These are summarized in Table 24-2. Several of these methods imply the modification of a collection. If the collection is read-only or of fixed size, then these methods will throw a **NotSupportedException**.

Objects are added to an **IList** collection by calling **Add( )**. Notice that **Add( )** takes an argument of type **object**. Since **object** is a base class for all types, any type of object can be stored in a non-generic collection. This includes the value types, because boxing and unboxing will automatically take place.

You can remove an element using **Remove( )** or **RemoveAt( )**. **Remove( )** removes the specified object. **RemoveAt( )** removes the object at a specified index. To empty the collection, call **Clear( )**.

Method	Description
int Add(object *obj*)	Adds *obj* into the invoking collection. Returns the index at which the object is stored.
void Clear( )	Deletes all elements from the invoking collection.
bool Contains(object *obj*)	Returns true if the invoking collection contains *obj*. Returns false if *obj* is not in the collection.
int IndexOf(object *obj*)	Returns the index of *obj* if *obj* is contained within the invoking collection. If *obj* is not found, –1 is returned.
void Insert(int *idx*, object *obj*)	Inserts *obj* at the index specified by *idx*. Elements at and below *idx* are moved down to make room for *obj*.
void Remove(object *obj*)	Removes the first occurrence of *obj* from the invoking collection. Elements at and below the removed element are moved up to close the gap.
void RemoveAt(int *idx*)	Removes the object at the index specified by *idx* from the invoking collection. Elements at and below *idx* are moved up to close the gap.

Table 24-2 The Methods Defined by **IList**

You can determine whether a collection contains a specific object by calling **Contains( )**. You can obtain the index of an object by called **IndexOf( )**. You can insert an element at a specific index by calling **Insert( )**.

IList defines the following properties:

```
bool IsFixedSize { get; }
bool IsReadOnly { get; }
```

If the collection is of fixed size, **IsFixedSize** is true. This means elements cannot be inserted or removed. If the collection is read-only, then **IsReadOnly** is true. This means the contents of the collection cannot be changed.

IList defines the following indexer:

```
object this[int idx] { get; set; }
```

You will use this indexer to get or set the value of an element. However, you cannot use it to add a new element to the collection. To add an element to a list, call **Add( )**. Once it is added, you can access the element through the indexer.

The IDictionary Interface

The **IDictionary** interface defines the behavior of a non-generic collection that maps unique keys to values. A key is an object that you use to retrieve a value at a later date. Thus, a collection that implements **IDictionary** stores key/value pairs. Once the pair is stored, you can retrieve it by using its key. **IDictionary** inherits **ICollection** and **IEnumerable**. The methods declared by **IDictionary** are summarized in Table 24-3. Several methods throw an **ArgumentNullException** if an attempt is made to specify a null key and null keys are not allowed.

To add a key/value pair to an **IDictionary** collection, use **Add( )**. Notice that the key and its value are specified separately. To remove an element, specify the key of the object in a call to **Remove( )**. To empty the collection, call **Clear( )**.

You can determine whether a collection contains a specific object by calling **Contains( )** with the key of the desired item. **GetEnumerator( )** obtains an enumerator compatible with an **IDictionary** collection. This enumerator operates on key/value pairs.

Method	Description
void Add(object *k*, object *v*)	Adds the key/value pair specified by *k* and *v* to the invoking collection. *k* must not be null.
void Clear( )	Removes all key/value pairs from the invoking collection.
bool Contains(object *k*)	Returns true if the invoking collection contains *k* as a key. Otherwise, returns false.
IDictionaryEnumerator GetEnumerator( )	Returns the enumerator for the invoking collection.
void Remove(object *k*)	Removes the entry whose key equals *k*.

TABLE 24-3 The Methods Defined by **IDictionary**

IDictionary defines the following properties:

Property	Description
bool IsFixedSize { get; }	Is true if the dictionary is of fixed size.
bool IsReadOnly { get; }	Is true if the dictionary is read-only.
ICollection Keys { get; }	Obtains a collection of the keys.
ICollection Values { get; }	Obtains a collection of the values.

Notice that the keys and values contained within the collection are available as separate lists through the **Keys** and **Values** properties.

IDictionary defines the following indexer:

object this[object *key*] { get; set; }

You can use this indexer to get or set the value of an element. You can also use it to add a new element to the collection. Notice that the "index" is not actually an index, but rather the key of the item.

IEnumerable, IEnumerator, and IDictionaryEnumerator

IEnumerable is the non-generic interface that a class must implement if it is to support enumerators. As explained, all of the non-generic collection classes implement **IEnumerable** because it is inherited by **ICollection**. The sole method defined by **IEnumerable** is **GetEnumerator( )**, which is shown here:

IEnumerator GetEnumerator( )

It returns the enumerator for the collection. Also, implementing **IEnumerable** allows the contents of a collection to be obtained by a **foreach** loop.

IEnumerator is the interface that defines the functionality of an enumerator. Using its methods, you can cycle through the contents of a collection. For collections that store key/value pairs (dictionaries), **GetEnumerator( )** returns an object of type **IDictionaryEnumerator**, rather than **IEnumerator**. **IDictionaryEnumerator** inherits **IEnumerator** and adds functionality to facilitate the enumeration of dictionaries.

IEnumerator defines the methods **MoveNext( )** and **Reset( )** and the **Current** property. The techniques needed to use them are described in detail later in this chapter. Briefly, **Current** holds the element currently being obtained. **MoveNext( )** moves to the next element. **Reset( )** restarts the enumeration from the beginning.

IComparer and IEqualityComparer

The **IComparer** interface defines a method called **Compare( )**, which defines the way two objects are compared. It is shown here:

int Compare(object *v1*, object *v2*)

It must return greater than zero if *v1* is greater than *v2*, less than zero if *v1* is less than *v2*, and zero if the two values are the same. This interface can be used to specify how the elements of a collection should be sorted.

IEqualityComparer defines these two methods:

bool Equals(object *obj1*, object *obj2*)

int GetHashCode(object *obj*)

Equals( ) must return true if **obj1** and **obj2** are equal. **GetHashCode( )** must return the hash code for *obj*.

The DictionaryEntry Structure

System.Collections defines one structure type called **DictionaryEntry**. Non-generic collections that hold key/value pairs store those pairs in a **DictionaryEntry** object. This structure defines the following two properties:

```
public object Key { get; set; }
public object Value { get; set; }
```

These properties are used to access the key or value associated with an entry. You can construct a **DictionaryEntry** object by using the following constructor:

```
public DictionaryEntry(object k, object v)
```

Here, *k* is the key and *v* is the value.

The Non-Generic Collection Classes

Now that you are familiar with the non-generic collection interfaces, we can examine the standard classes that implement them. With the exception of **BitArray**, described later, the non-generic collection classes are summarized here:

Class	Description
ArrayList	A dynamic array. This is an array that can grow as needed.
Hashtable	A hash table for key/value pairs.
Queue	A first-in, first-out list.
SortedList	A sorted list of key/value pairs.
Stack	A first-in, last-out list.

The following sections examine these collection classes and illustrate their use.

ArrayList

The **ArrayList** class supports dynamic arrays, which can grow or shrink as needed. In C#, standard arrays are of a fixed length, which cannot be changed during program execution. This means you must know in advance how many elements an array will hold. But sometimes you may not know until runtime precisely how large an array you will need. To handle this situation, use **ArrayList**. An **ArrayList** is a variable-length array of object references that can dynamically increase or decrease in size. An **ArrayList** is created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array can be shrunk. **ArrayList** is currently in wide use in existing code. For this reason, it is examined in depth here. However, many of the same techniques that apply to **ArrayList** apply to the other collections as well, including the generic collections.

ArrayList implements **ICollection**, **IList**, **IEnumerable**, and **ICloneable**. **ArrayList** has the constructors shown here:

```
public ArrayList( )
public ArrayList(ICollection c)
public ArrayList(int capacity)
```

The first constructor builds an empty **ArrayList** with an initial capacity of zero. The second constructor builds an **ArrayList** that is initialized with the elements specified by *c* and has an initial capacity equal to the number of elements. The third constructor builds an array list that has the specified initial *capacity*. The capacity is the size of the underlying array that is used to store the elements. The capacity grows automatically as elements are added to an **ArrayList**.

In addition to the methods defined by the interfaces that it implements, **ArrayList** defines several methods of its own. Some of the more commonly used ones are shown in Table 24-4. An **ArrayList** can be sorted by calling **Sort( )**. Once sorted, it can be efficiently searched by **BinarySearch( )**. The contents of an **ArrayList** can be reversed by calling **Reverse( )**.

ArrayList supports several methods that operate on a range of elements within a collection. You can insert another collection into an **ArrayList** by calling **InsertRange( )**. You can remove a range by calling **RemoveRange( )**. You can overwrite a range within an **ArrayList** with the elements of another collection by calling **SetRange( )**. You can also sort or search a range rather than the entire collection.

By default, an **ArrayList** is not synchronized. To obtain a synchronized wrapper around a collection, call **Synchronized( )**.

Method	Description
public virtual void AddRange(ICollection *c*)	Adds the elements in *c* to the end of the invoking **ArrayList**.
public virtual int BinarySearch(object *v*)	Searches the invoking collection for the value passed in *v*. The index of the matching element is returned. If the value is not found, a negative value is returned. The invoking list must be sorted.
public virtual int BinarySearch(object *v*, IComparer *comp*)	Searches the invoking collection for the value passed in *v* using the comparison object specified by *comp*. The index of the matching element is returned. If the value is not found, a negative value is returned. The invoking list must be sorted.
public virtual int BinarySearch(int *startIdx*, int *count*, object *v*, IComparer *comp*)	Searches the invoking collection for the value passed in *v* using the comparison object specified by *comp*. The search begins at *startIdx* and runs for *count* elements. The index of the matching element is returned. If the value is not found, a negative value is returned. The invoking list must be sorted.
public virtual void CopyTo(Array *ar*)	Copies the contents of the invoking collection to the array specified by *ar*, which must be a one-dimensional array compatible with the type of the elements in the collection.
public virtual void CopyTo(Array *ar*, int *startIdx*)	Copies the contents of the invoking collection to the array specified by *ar*, beginning at *startIdx*. The array must be a one-dimensional array compatible with the type of the elements in the collection.

TABLE 24-4 Several Commonly Used Methods Defined by **ArrayList**

Method	Description
public virtual void CopyTo(int *srcIdx*, Array *ar*, int *destIdx*, int *count*)	Copies a portion of the invoking collection, beginning at *srcIdx* and running for *count* elements, to the array specified by *ar,* beginning at *destIdx. ar* must be a one-dimensional array compatible with the type of the elements in the collection.
public static ArrayList FixedSize(ArrayList *ar*)	Wraps *ar* in a fixed-size **ArrayList** and returns the result.
public virtual ArrayList GetRange(int *idx*, int *count*)	Returns a portion of the invoking **ArrayList**. The range returned begins at *idx* and runs for *count* elements. The returned object refers to the same elements as the invoking object.
public virtual int IndexOf(object *v*)	Returns the index of the first occurrence of *v* in the invoking collection. Returns –1 if *v* is not found.
public virtual void InsertRange(int *startIdx*, ICollection *c*)	Inserts the elements of *c* into the invoking collection, starting at the index specified by *startIdx.*
public virtual int LastIndexOf(object *v*)	Returns the index of the last occurrence of *v* in the invoking collection. Returns –1 if *v* is not found.
public static ArrayList ReadOnly(ArrayList *ar*)	Wraps *ar* in a read-only **ArrayList** and returns the result.
public virtual void RemoveRange(int *idx*, int *count*)	Removes *count* elements from the invoking collection, beginning at *idx.*
public virtual void Reverse( )	Reverses the contents of the invoking collection.
public virtual void Reverse(int *startIdx*, int *count*)	Reverses *count* elements of the invoking collection, beginning at *startIdx.*
public virtual void SetRange(int *startIdx*, ICollection *c*)	Replaces elements within the invoking collection, beginning at *startIdx,* with those specified by *c.*
public virtual void Sort( )	Sorts the collection into ascending order.
public virtual void Sort(IComparer *comp*)	Sorts the collection using the specified comparison object. If *comp* is null, the default comparison for each object is used.
public virtual void Sort(int *startIdx*, int *count*, IComparer *comp*)	Sorts a portion of the collection using the specified comparison object. The sort begins at *startIdx* and runs for *count* elements. If *comp* is null, the default comparison for each object is used.
public static ArrayList Synchronized(ArrayList *list*)	Returns a synchronized version of the invoking **ArrayList**.
public virtual object[ ] ToArray( )	Returns an array that contains copies of the elements of the invoking object.

Table 24-4 Several Commonly Used Methods Defined by **ArrayList** *(continued)*

Method	Description
public virtual Array ToArray(Type *type*)	Returns an array that contains copies of the elements of the invoking object. The type of the elements in the array are specified by *type*.
public virtual void TrimToSize( )	Sets **Capacity** to **Count**.

TABLE 24-4 Several Commonly Used Methods Defined by **ArrayList** *(continued)*

In addition to those properties defined by the interfaces that it implements, **ArrayList** adds **Capacity**, shown here:

public virtual int Capacity { get; set; }

Capacity gets or sets the capacity of the invoking **ArrayList**. The capacity is the number of elements that can be held before the **ArrayList** must be enlarged. As mentioned, an **ArrayList** grows automatically, so it is not necessary to set the capacity manually. However, for efficiency reasons, you might want to set the capacity when you know in advance how many elements the list will contain. This prevents the overhead associated with the allocation of more memory.

Conversely, if you want to reduce the size of the array that underlies an **ArrayList**, you can set **Capacity** to a smaller value. However, this value must not be less than **Count**. Recall that **Count** is a property defined by **ICollection** that holds the number of objects currently stored in a collection. Attempting to set **Capacity** to a value less than **Count** causes an **ArgumentOutOfRangeException** to be generated. To obtain an **ArrayList** that is precisely as large as the number of items that it is currently holding, set **Capacity** equal to **Count**. You can also call **TrimToSize( )**.

The following program demonstrates **ArrayList**. It creates an **ArrayList** and then adds characters to it. The list is then displayed. Some of the elements are removed, and the list is displayed again. Next, more elements are added, forcing the capacity of the list to be increased. Finally, the contents of elements are changed.

```
// Demonstrate ArrayList.

using System;
using System.Collections;

class ArrayListDemo {
  static void Main() {
    // Create an array list.
    ArrayList al = new ArrayList();

    Console.WriteLine("Initial number of elements: " +
                      al.Count);

    Console.WriteLine();

    Console.WriteLine("Adding 6 elements");
    // Add elements to the array list
    al.Add('C');
```

```
    al.Add('A');
    al.Add('E');
    al.Add('B');
    al.Add('D');
    al.Add('F');

    Console.WriteLine("Number of elements: " +
                      al.Count);

    // Display the array list using array indexing.
    Console.Write("Current contents: ");
    for(int i=0; i < al.Count; i++)
      Console.Write(al[i] + " ");
    Console.WriteLine("\n");

    Console.WriteLine("Removing 2 elements");
    // Remove elements from the array list.
    al.Remove('F');
    al.Remove('A');

    Console.WriteLine("Number of elements: " +
                      al.Count);

    // Use foreach loop to display the list.
    Console.Write("Contents: ");
    foreach(char c in al)
      Console.Write(c + " ");
    Console.WriteLine("\n");

    Console.WriteLine("Adding 20 more elements");
    // Add enough elements to force al to grow.
    for(int i=0; i < 20; i++)
      al.Add((char)('a' + i));
    Console.WriteLine("Current capacity: " +
                      al.Capacity);
    Console.WriteLine("Number of elements after adding 20: " +
                      al.Count);
    Console.Write("Contents: ");
    foreach(char c in al)
      Console.Write(c + " ");
    Console.WriteLine("\n");

    // Change contents using array indexing.
    Console.WriteLine("Change first three elements");
    al[0] = 'X';
    al[1] = 'Y';
    al[2] = 'Z';
    Console.Write("Contents: ");
    foreach(char c in al)
      Console.Write(c + " ");
    Console.WriteLine();
  }
}
```

The output from this program is shown here:

```
Initial number of elements: 0

Adding 6 elements
Number of elements: 6
Current contents: C A E B D F

Removing 2 elements
Number of elements: 4
Contents: C E B D

Adding 20 more elements
Current capacity: 32
Number of elements after adding 20: 24
Contents: C E B D a b c d e f g h i j k l m n o p q r s t

Change first three elements
Contents: X Y Z D a b c d e f g h i j k l m n o p q r s t
```

Sorting and Searching an ArrayList An **ArrayList** can be sorted by **Sort( )**. Once sorted, it can be efficiently searched by **BinarySearch( )**. The following program demonstrates these methods:

```
// Sort and search an ArrayList.

using System;
using System.Collections;

class SortSearchDemo {
  static void Main() {
    // Create an array list.
    ArrayList al = new ArrayList();

    // Add elements to the array list.
    al.Add(55);
    al.Add(43);
    al.Add(-4);
    al.Add(88);
    al.Add(3);
    al.Add(19);

    Console.Write("Original contents: ");
    foreach(int i in al)
      Console.Write(i + " ");
    Console.WriteLine("\n");

    // Sort
    al.Sort();

    // Use foreach loop to display the list.
    Console.Write("Contents after sorting: ");
    foreach(int i in al)
      Console.Write(i + " ");
    Console.WriteLine("\n");
```

```
    Console.WriteLine("Index of 43 is " +
                      al.BinarySearch(43));
  }
}
```

The output is shown here:

```
Original contents: 55 43 -4 88 3 19

Contents after sorting: -4 3 19 43 55 88

Index of 43 is 3
```

Although an **ArrayList** can store objects of any type within the same list, when sorting or searching a list, it is necessary for those objects to be comparable. For example, the preceding program would have generated an exception if the list had included a string. (It is possible to create custom comparison methods that would allow the comparison of strings and integers, however. Custom comparators are discussed later in this chapter.)

Obtaining an Array from an ArrayList When working with **ArrayList**, you will sometimes want to obtain an actual array that contains the contents of the list. You can do this by calling **ToArray( )**. There are several reasons why you might want to convert a collection into an array. Here are two: You may want to obtain faster processing times for certain operations, or you might need to pass an array to a method that is not overloaded to accept a collection. Whatever the reason, converting an **ArrayList** to an array is a trivial matter, as the following program shows:

```
// Convert an ArrayList into an array.

using System;
using System.Collections;

class ArrayListToArray {
  static void Main() {
    ArrayList al = new ArrayList();

    // Add elements to the array list.
    al.Add(1);
    al.Add(2);
    al.Add(3);
    al.Add(4);

    Console.Write("Contents: ");
    foreach(int i in al)
      Console.Write(i + " ");
    Console.WriteLine();

    // Get the array.
    int[] ia = (int[]) al.ToArray(typeof(int));
    int sum = 0;

    // Sum the array.
    for(int i=0; i<ia.Length; i++)
      sum += ia[i];
```

```
    Console.WriteLine("Sum is: " + sum);
  }
}
```

The output from the program is shown here:

```
Contents: 1 2 3 4
Sum is: 10
```

The program begins by creating a collection of integers. Next, **ToArray( )** is called with the type specified as **int**. This causes an array of integers to be created. Since the return type of **ToArray( )** is **Array**, the contents of the array must still be cast to **int[ ]**. (Recall that **Array** is the base type of all C# arrays.) Finally, the values are summed.

Hashtable

Hashtable creates a collection that uses a hash table for storage. As most readers will know, a *hash table* stores information using a mechanism called *hashing*. In hashing, the informational content of a key is used to determine a unique value, called its *hash code*. The hash code is then used as the index at which the data associated with the key is stored in the table. The transformation of the key into its hash code is performed automatically—you never see the hash code itself. The advantage of hashing is that it allows the execution time of lookup, retrieve, and set operations to remain near constant, even for large sets. **Hashtable** implements the **IDictionary**, **ICollection**, **IEnumerable**, **ISerializable**, **IDeserializationCallback**, and **ICloneable** interfaces.

Hashtable defines many constructors, including these frequently used ones:

public Hashtable( )
public Hashtable(IDictionary *c*)
public Hashtable(int *capacity*)
public Hashtable(int *capacity*, float *fillRatio*)

The first form constructs a default **Hashtable**. The second form initializes the **Hashtable** by using the elements of *c*. The third form initializes the capacity of the **Hashtable** to *capacity*. The fourth form initializes both the capacity and fill ratio. The fill ratio (also called the *load factor*) must be between 0.1 and 1.0, and it determines how full the hash table can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the table multiplied by its fill ratio, the table is expanded. For constructors that do not take a fill ratio, 1.0 is used.

In addition to the methods defined by the interfaces that it implements, **Hashtable** also defines several methods of its own. Some commonly used ones are shown in Table 24-5. To determine if a **Hashtable** contains a key, call **ContainsKey( )**. To see if a specific value is stored, call **ContainsValue( )**. To enumerate the contents of a **Hashtable**, obtain an **IDictionaryEnumerator** by calling **GetEnumerator( )**. Recall that **IDictionaryEnumerator** is used to enumerate the contents of a collection that stores key/value pairs.

The public properties available in **Hashtable** are those defined by the interfaces that it implements. Two especially important ones are **Keys** and **Values** because they let you obtain a collection of a **Hashtable**'s keys or values. They are specified by **IDictionary** and are shown here:

public virtual ICollection Keys { get; }
public virtual ICollection Values { get; }

Method	Description
public virtual bool ContainsKey(object *k*)	Returns true if *k* is a key in the invoking **Hashtable**. Returns false otherwise.
public virtual bool ContainsValue(object *v*)	Returns true if *v* is a value in the invoking **Hashtable**. Returns false otherwise.
public virtual IDictionaryEnumerator GetEnumerator( )	Returns an **IDictionaryEnumerator** for the invoking **Hashtable**.
public static Hashtable Synchronized(Hashtable *ht*)	Returns a synchronized version of the **Hashtable** passed in *ht*.

TABLE 24-5 Several Commonly Used Methods Defined by **Hashtable**

Because **Hashtable** does not maintain an ordered collection, there is no specific order to the collection of keys or values obtained. **Hashtable** also has two protected properties: **EqualityComparer** and **KeyComparer**. Two other properties called **hcp** and **comparer** are flagged as obsolete.

Hashtable stores key/value pairs in the form of a **DictionaryEntry** structure, but most of the time, you won't be aware of it directly because the properties and methods work with keys and values individually. For example, when you add an element to a **Hashtable**, you call **Add( )**, which takes two arguments: the key and the value.

It is important to note that **Hashtable** does not guarantee the order of its elements. This is because the process of hashing does not usually lend itself to the creation of sorted tables.

Here is an example that demonstrates **Hashtable**:

```
// Demonstrate Hashtable.

using System;
using System.Collections;

class HashtableDemo {
  static void Main() {
    // Create a hash table.
    Hashtable ht = new Hashtable();

    // Add elements to the table.
    ht.Add("house", "Dwelling");
    ht.Add("car", "Means of transport");
    ht.Add("book", "Collection of printed words");
    ht.Add("apple", "Edible fruit");

    // Can also add by using the indexer.
    ht["tractor"] = "Farm implement";

    // Get a collection of the keys.
    ICollection c = ht.Keys;

    // Use the keys to obtain the values.
    foreach(string str in c)
      Console.WriteLine(str + ": " + ht[str]);
  }
}
```

The output from this program is shown here:

```
book: Collection of printed words
tractor: Farm implement
apple: Edible fruit
house: Dwelling
car: Means of transport
```

As the output shows, the key/value pairs are not stored in sorted order. Notice how the contents of the hash table **ht** were obtained and displayed. First, a collection of the keys was retrieved by the **Keys** property. Each key was then used to index **ht**, yielding the value associated with each key. Remember, the indexer defined by **IDictionary** and implemented by **Hashtable** uses a key as the index.

SortedList

SortedList creates a collection that stores key/value pairs in sorted order, based on the value of the keys. **SortedList** implements the **IDictionary**, **ICollection**, **IEnumerable**, and **ICloneable** interfaces.

SortedList has several constructors, including those shown here:

public SortedList( )
public SortedList(IDictionary *c*)
public SortedList(int *capacity*)
public SortedList(IComparer *comp*)

The first constructor builds an empty collection with an initial capacity of zero. The second constructor builds a **SortedList** that is initialized with the elements of *c* and has an initial capacity equal to the number of elements. The third constructor builds an empty **SortedList** that has the initial capacity specified by *capacity.* The capacity is the size of the underlying array that is used to store the elements. The fourth form lets you specify a comparison method that will be used to compare the object contained in the list. This form creates an empty collection with an initial capacity of zero.

The capacity of a **SortedList** grows automatically as needed when elements are added to the list. When the current capacity is exceeded, the capacity is increased. The advantage of specifying a capacity when creating a **SortedList** is that you can prevent or minimize the overhead associated with resizing the collection. Of course, it makes sense to specify an initial capacity only if you have some idea of how many elements will be stored.

In addition to the methods defined by the interfaces that it implements, **SortedList** also defines several methods of its own. Some of the most commonly used ones are shown in Table 24-6. To determine if a **SortedList** contains a key, call **ContainsKey( )**. To see if a specific value is stored, call **ContainsValue( )**. To enumerate the contents of a **SortedList**, obtain an **IDictionaryEnumerator** by calling **GetEnumerator( )**. Recall that **IDictionaryEnumerator** is used to enumerate the contents of a collection that stores key/value pairs. You can obtain a synchronized wrapper around a **SortedList** by calling **Synchronized( )**.

There are various ways to set or obtain a value or key. To obtain the value associated with a specific index, call **GetByIndex( )**. To set a value given its index, call **SetByIndex( )**. You can retrieve the key associated with a specific index by calling **GetKey( )**. To obtain a list of all the keys, use **GetKeyList( )**. To get a list of all the values, use **GetValueList( )**. You can obtain the index of a key by calling **IndexOfKey( )** and the index of a value by calling **IndexOfValue( )**. Of course, **SortedList** also supports the indexer defined by **IDictionary** that lets you set or obtain a value given its key.

Method	Description
public virtual bool ContainsKey(object *k*)	Returns true if *k* is a key in the invoking **SortedList**. Returns false otherwise.
public virtual bool ContainsValue(object *v*)	Returns true if *v* is a value in the invoking **SortedList**. Returns false otherwise.
public virtual object GetByIndex(int *idx*)	Returns the value at the index specified by *idx*.
public virtual IDictionaryEnumerator GetEnumerator( )	Returns an **IDictionaryEnumerator** for the invoking **SortedList**.
public virtual object GetKey(int *idx*)	Returns the value of the key at the index specified by *idx*.
public virtual IList GetKeyList( )	Returns an **IList** collection of the keys in the invoking **SortedList**.
public virtual IList GetValueList( )	Returns an **IList** collection of the values in the invoking **SortedList**.
public virtual int IndexOfKey(object *k*)	Returns the index of the key specified by *k*. Returns –1 if the key is not in the list.
public virtual int IndexOfValue(object *v*)	Returns the index of the first occurrence of the value specified by *v*. Returns –1 if the value is not in the list.
public virtual void SetByIndex(int *idx*, object *v*)	Sets the value at the index specified by *idx* to the value passed in *v*.
public static SortedList Synchronized(SortedList *sl*)	Returns a synchronized version of the **SortedList** passed in *sl*.
public virtual void TrimToSize( )	Sets **Capacity** to **Count**.

TABLE 24-6 Several Commonly Used Methods Defined by **SortedList**

The public properties available in **SortedList** are those defined by the interfaces that it implements. As is the case with **Hashtable**, two especially important properties are **Keys** and **Values** because they let you obtain a read-only collection of a **SortedList**'s keys or values. They are specified by **IDictionary** and are shown here:

```
public virtual ICollection Keys { get; }
public virtual ICollection Values { get; }
```

The order of the keys and values reflects that of the **SortedList**.

Like **Hashtable**, a **SortedList** stores key/value pairs in the form of a **DictionaryEntry** structure, but you will usually access the keys and values individually using the methods and properties defined by **SortedList**.

The following program demonstrates **SortedList**. It reworks and expands the **Hashtable** demonstration program from the previous section, substituting **SortedList**. When you examine the output, you will see that the **SortedList** version is sorted by key.

```
// Demonstrate a SortedList.

using System;
using System.Collections;
```

```
class SLDemo {
  static void Main() {
    // Create a sorted SortedList.
    SortedList sl = new SortedList();

    // Add elements to the table.
    sl.Add("house", "Dwelling");
    sl.Add("car", "Means of transport");
    sl.Add("book", "Collection of printed words");
    sl.Add("apple", "Edible fruit");

    // Can also add by using the indexer.
    sl["tractor"] = "Farm implement";

    // Get a collection of the keys.
    ICollection c = sl.Keys;

    // Use the keys to obtain the values.
    Console.WriteLine("Contents of list via indexer.");
    foreach(string str in c)
      Console.WriteLine(str + ": " + sl[str]);

    Console.WriteLine();

    // Display list using integer indexes.
    Console.WriteLine("Contents by integer indexes.");
    for(int i=0; i < sl.Count; i++)
      Console.WriteLine(sl.GetByIndex(i));

    Console.WriteLine();

    // Show integer indexes of entries.
    Console.WriteLine("Integer indexes of entries.");
    foreach(string str in c)
      Console.WriteLine(str + ": " + sl.IndexOfKey(str));
  }
}
```

The output is shown here:

```
Contents of list via indexer.
apple: Edible fruit
book: Collection of printed words
car: Means of transport
house: Dwelling
tractor: Farm implement

Contents by integer indexes.
Edible fruit
Collection of printed words
Means of transport
Dwelling
Farm implement
```

```
Integer indexes of entries.
apple: 0
book: 1
car: 2
house: 3
tractor: 4
```

Stack

As most readers know, a stack is a first-in, last-out list. To visualize a stack, imagine a stack of plates on a table. The first plate put down is the last one to be picked up. The stack is one of the most important data structures in computing. It is frequently used in system software, compilers, and AI-based backtracking routines, to name just a few examples.

The collection class that supports a stack is called **Stack**. It implements the **ICollection**, **IEnumerable**, and **ICloneable** interfaces. **Stack** is a dynamic collection that grows as needed to accommodate the elements it must store. Each time the capacity must be increased, the capacity is doubled.

Stack defines the following constructors:

public Stack( )
public Stack(int *capacity*)
public Stack(ICollection *c*)

The first form creates an empty stack. The second form creates an empty stack with the initial capacity specified by *capacity.* The third form creates a stack that contains the elements of the collection specified by *c* and an initial capacity equal to the number of elements.

In addition to the methods defined by the interfaces that it implements, **Stack** defines the methods shown in Table 24-7. In general, here is how you use **Stack**. To put an object on the top of the stack, call **Push( )**. To remove and return the top element, call **Pop( )**. You can use **Peek( )** to return, but not remove, the top object. An **InvalidOperationException** is thrown if you call **Pop( )** or **Peek( )** when the invoking stack is empty.

Method	Description
public virtual void Clear( )	Sets **Count** to zero, which effectively clears the stack.
public virtual bool Contains(object *v*)	Returns true if *v* is on the invoking stack. If *v* is not found, false is returned.
public virtual object Peek( )	Returns the element on the top of the stack, but does not remove it.
public virtual object Pop( )	Returns the element on the top of the stack, removing it in the process.
public virtual void Push(object *v*)	Pushes *v* onto the stack.
public static Stack Synchronized(Stack *stk*)	Returns a synchronized version of the **Stack** passed in *stk.*
public virtual object[ ] ToArray( )	Returns an array that contains copies of the elements of the invoking stack.

Table 24-7 The Methods Defined by **Stack**

Here is an example that creates a stack, pushes several integers onto it, and then pops them off again:

```
// Demonstrate the Stack class.

using System;
using System.Collections;

class StackDemo {
  static void ShowPush(Stack st, int a) {
    st.Push(a);
    Console.WriteLine("Push(" + a + ")");

    Console.Write("stack: ");
    foreach(int i in st)
      Console.Write(i + " ");

    Console.WriteLine();
  }

  static void ShowPop(Stack st) {
    Console.Write("Pop -> ");
    int a = (int) st.Pop();
    Console.WriteLine(a);

    Console.Write("stack: ");
    foreach(int i in st)
      Console.Write(i + " ");

    Console.WriteLine();
  }

  static void Main() {
    Stack st = new Stack();

    foreach(int i in st)
      Console.Write(i + " ");

    Console.WriteLine();

    ShowPush(st, 22);
    ShowPush(st, 65);
    ShowPush(st, 91);
    ShowPop(st);
    ShowPop(st);
    ShowPop(st);

    try {
      ShowPop(st);
    } catch (InvalidOperationException) {
      Console.WriteLine("Stack empty.");
```

```
    }
  }
}
```

Here's the output produced by the program. Notice how the exception handler for **InvalidOperationException** manages a stack underflow.

```
Push(22)
stack: 22
Push(65)
stack: 65 22
Push(91)
stack: 91 65 22
Pop -> 91
stack: 65 22
Pop -> 65
stack: 22
Pop -> 22
stack:
Pop -> Stack empty.
```


Queue

Another familiar data structure is the queue, which is a first-in, first-out list. That is, the first item put in a queue is the first item retrieved. Queues are common in real life. For example, lines at a bank or fast-food restaurant are queues. In programming, queues are used to hold such things as the currently executing processes in the system, a list of pending database transactions, or data packets received over the Internet. They are also often used in simulations.

The collection class that supports a queue is called **Queue**. It implements the **ICollection**, **IEnumerable**, and **ICloneable** interfaces. **Queue** is a dynamic collection that grows as needed to accommodate the elements it must store. When more room is needed, the size of the queue is increased by a growth factor, which, by default, is 2.0.

Queue defines the following constructors:

public Queue( )
public Queue (int *capacity*)
public Queue (int *capacity*, float *growFact*)
public Queue (ICollection *c*)

The first form creates an empty queue with an initial capacity of 32 and uses the default growth factor of 2.0. The second form creates an empty queue with the initial capacity specified by *capacity* and a growth factor of 2.0. The third form allows you to specify a growth factor in *growFact* (which must be between 1.0 and 10.0). The fourth form creates a queue that contains the elements of the collection specified by *c*, and an initial capacity equal to the number of elements. In this form, the default growth factor of 2.0 is used.

In addition to the methods defined by the interfaces that it implements, **Queue** defines the methods shown in Table 24-8. In general, here is how you use **Queue**. To put an object in the queue, call **Enqueue( )**. To remove and return the object at the front of the queue, call **Dequeue( )**. You can use **Peek( )** to return, but not remove, the next object. An **InvalidOperationException** is thrown if you call **Dequeue( )** or **Peek( )** when the invoking queue is empty.

Method	Description
public virtual void Clear( )	Sets **Count** to zero, which effectively clears the queue.
public virtual bool Contains(object *v*)	Returns true if *v* is in the invoking queue. If *v* is not found, false is returned.
public virtual object Dequeue( )	Returns the object at the front of the invoking queue. The object is removed in the process.
public virtual void Enqueue(object *v*)	Adds *v* to the end of the queue.
public virtual object Peek( )	Returns the object at the front of the invoking queue, but does not remove it.
public static Queue Synchronized(Queue *q*)	Returns a synchronized version of *q*.
public virtual object[ ] ToArray( )	Returns an array that contains copies of the elements of the invoking queue.
public virtual void TrimToSize( )	Sets **Capacity** to **Count**.

TABLE 24-8 The Methods Defined by **Queue**

Here is an example that demonstrates **Queue**:

```
// Demonstrate the Queue class.

using System;
using System.Collections;

class QueueDemo {
  static void ShowEnq(Queue q, int a) {
    q.Enqueue(a);
    Console.WriteLine("Enqueue(" + a + ")");

    Console.Write("queue: ");
    foreach(int i in q)
      Console.Write(i + " ");

    Console.WriteLine();
  }

  static void ShowDeq(Queue q) {
    Console.Write("Dequeue -> ");
    int a = (int) q.Dequeue();
    Console.WriteLine(a);

    Console.Write("queue: ");
    foreach(int i in q)
      Console.Write(i + " ");

    Console.WriteLine();
  }

  static void Main() {
```

```
    Queue q = new Queue();

    foreach(int i in q)
      Console.Write(i + " ");

    Console.WriteLine();

    ShowEnq(q, 22);
    ShowEnq(q, 65);
    ShowEnq(q, 91);
    ShowDeq(q);
    ShowDeq(q);
    ShowDeq(q);

    try {
      ShowDeq(q);
    } catch (InvalidOperationException) {
      Console.WriteLine("Queue empty.");
    }
  }
}
```

The output is shown here:

```
Enqueue(22)
queue: 22
Enqueue(65)
queue: 22 65
Enqueue(91)
queue: 22 65 91
Dequeue -> 22
queue: 65 91
Dequeue -> 65
queue: 91
Dequeue -> 91
queue:
Dequeue -> Queue empty.
```

Storing Bits with BitArray

The **BitArray** class supports a collection of bits. Because it stores bits rather than objects, **BitArray** has capabilities different from those of the other collections. However, it still supports the basic collection underpinning by implementing **ICollection** and **IEnumerable**. It also implements **ICloneable**.

BitArray defines several constructors. You can construct a **BitArray** from an array of Boolean values using this constructor:

public BitArray(bool[ ] *bits*)

In this case, each element of *bits* becomes a bit in the collection. Thus, each bit in the collection corresponds to an element of *bits*. Furthermore, the ordering of the elements of *bits* and the bits in the collection are the same.

You can create a **BitArray** from an array of bytes using this constructor:

public BitArray(byte[ ] *bits*)

Here, the bit pattern in *bits* becomes the bits in the collection, with *bits*[0] specifying the first 8 bits, *bits*[1] specifying the second 8 bits, and so on. In similar fashion, you can construct a **BitArray** from an array of **int**s using this constructor:

public BitArray(int[ ] *bits*)

In this case, *bits*[0] specifies the first 32 bits, *bits*[1] specifies the second 32 bits, and so on.

You can create a **BitArray** of a specific size using this constructor:

public BitArray(int *size*)

Here, *size* specifies the number of bits. The bits in the collection are initialized to **false**. To specify a size and initial value of the bits, use the following constructor:

public BitArray(int *size*, bool *v*)

In this case, all bits in the collection will be set to the value passed in *v*.

Finally, you can create a new **BitArray** from an existing one by using this constructor:

public BitArray(BitArray *bits*)

The new object will contain the same collection of bits as *bits*, but the two collections will be otherwise separate.

BitArrays can be indexed. Each index specifies an individual bit, with an index of zero indicating the low-order bit.

In addition to the methods specified by the interfaces that it implements, **BitArray** defines the methods shown in Table 24-9. Notice that **BitArray** does not supply a **Synchronized( )** method. Thus, a synchronized wrapper is not available, and the **IsSynchronized** property is always false. However, you can control access to a **BitArray** by synchronizing on the object provided by **SyncRoot**.

Method	Description
public BitArray And(BitArray *ba*)	ANDs the bits of the invoking object with those specified by *ba* and returns a **BitArray** that contains the result.
public bool Get(int *idx*)	Returns the value of the bit at the index specified by *idx*.
public BitArray Not( )	Performs a bitwise, logical NOT on the invoking collection and returns a **BitArray** that contains the result.
public BitArray Or(BitArray *ba*)	ORs the bits of the invoking object with those specified by *ba* and returns a **BitArray** that contains the result.
public void Set(int *idx*, bool *v*)	Sets the bit at the index specified by *idx* to *v*.
public void SetAll(bool *v*)	Sets all bits to *v*.
public BitArray Xor(BitArray *ba*)	XORs the bits of the invoking object with those specified by *ba* and returns a **BitArray** that contains the result.

TABLE 24-9 The Methods Defined by **BitArray**

To the properties specified by the interfaces that it implements, **BitArray** adds **Length**, which is shown here:

public int Length { get; set; }

Length sets or obtains the number of bits in the collection. Thus, **Length** gives the same value as does the standard **Count** property, which is defined for all collections. However, **Count** is read-only, but **Length** is not. Thus, **Length** can be used to change the size of a **BitArray**. If you shorten a **BitArray**, bits are truncated from the high-order end. If you lengthen a **BitArray**, false bits are added to the high-order end.

BitArray defines the following indexer:

public bool this[int *idx*] { get; set; }

You can use this indexer to get or set the value of an element.

Here is an example that demonstrates **BitArray**:

```
// Demonstrate BitArray.

using System;
using System.Collections;

class BADemo {
  public static void ShowBits(string rem,
                        BitArray bits) {
    Console.WriteLine(rem);
    for(int i=0; i < bits.Count; i++)
      Console.Write("{0, -6} ", bits[i]);
    Console.WriteLine("\n");
  }

  static void Main() {
    BitArray ba = new BitArray(8);
    byte[] b = { 67 };
    BitArray ba2 = new BitArray(b);

    ShowBits("Original contents of ba:", ba);

    ba = ba.Not();

    ShowBits("Contents of ba after Not:", ba);

    ShowBits("Contents of ba2:", ba2);

    BitArray ba3 = ba.Xor(ba2);

    ShowBits("Result of ba XOR ba2:", ba3);
  }
}
```

The output is shown here:

```
Original contents of ba:
False  False  False  False  False  False  False  False
```

```
Contents of ba after Not:
True   True   True   True   True   True   True   True

Contents of ba2:
True   True   False  False  False  False  True   False

Result of ba XOR ba2:
False  False  True   True   True   True   False  True
```

The Specialized Collections

The .NET Framework provides some specialized collections that are optimized to work on a specific type of data or in a specific way. These non-generic collection classes are defined inside the **System.Collections.Specialized** namespace. They are synopsized in the following table:

Specialized Collection	Description
CollectionsUtil	Contains factory methods that create collections.
HybridDictionary	A collection that uses a **ListDictionary** to store key/value pairs when there are few elements in the collection. When the collection grows beyond a certain size, a **Hashtable** is automatically used to store the elements.
ListDictionary	A collection that stores key/value pairs in a linked list. It is recommended only for small collections.
NameValueCollection	A sorted collection of key/value pairs in which both the key and value are of type **string**.
OrderedDictionary	A collection of key/value pairs that can be indexed.
StringCollection	A collection optimized for storing strings.
StringDictionary	A hash table of key/value pairs in which both the key and the value are of type **string**.

System.Collections also defines three abstract base classes, **CollectionBase**, **ReadOnlyCollectionBase**, and **DictionaryBase**, which can be inherited and used as a starting point for developing custom specialized collections.

The Generic Collections

The addition of generics greatly expanded the Collections API, essentially doubling the amount of collection classes and interfaces. The generic collections are declared in the **System.Collections.Generic** namespace. In many cases, the generic collection classes are simply generic equivalents of the non-generic classes discussed earlier. However, the correspondence is not one-to-one. For example, there is a generic collection called **LinkedList** that implements a doubly linked list, but no non-generic equivalent. In some cases, parallel functionality exists between the generic and non-generic classes, but the names differ. For example, the generic version of **ArrayList** is called **List**, and the generic version of **HashTable**

is called **Dictionary**. Also, the specific contents of the various interfaces and classes contain minor reorganizations, with some functionality shifting from one interface to another, for example. However, overall, if you understand the non-generic collections, then you can easily use the generic collections.

In general, the generic collections work in the same way as the non-generic collections with the exception that a generic collection is type-safe. Thus, a generic collection can store only items that are compatible with its type argument. Therefore, if you want a collection that is capable of storing unrelated, mixed types, you should use one of the non-generic classes. However, for all cases in which a collection is storing only one type of object, then a generic collection is now your best choice.

The generic collections are defined by a set of interfaces and the classes that implement those interfaces. Each is described by the following sections.

The Generic Interfaces

System.Collections.Generic defines a number of generic interfaces, all of which parallel their corresponding non-generic counterparts. The generic interfaces are summarized in Table 24-10.

The ICollection<T> Interface

The **ICollection<T>** interface defines those features that all generic collections have in common. It inherits the **IEnumerable** and **IEnumerable<T>** interfaces. **ICollection<T>** is the generic version of the non-generic **ICollection** interface. However, there are some differences between the two.

ICollection<T> defines the following properties:

```
int Count { get; }

bool IsReadOnly { get; }
```

Count contains the number of items currently held in the collection. **IsReadOnly** is true if the collection is read-only. It is false if the collection is read/write.

Interface	Description
ICollection<T>	Defines the foundational features for the generic collections.
IComparer<T>	Defines the generic **Compare( )** method that performs a comparison on objects stored in a collection.
IDictionary<TK, TV>	Defines a generic collection that consists of key/value pairs.
IEnumerable<T>	Defines the generic **GetEnumerator( )** method, which supplies the enumerator for a collection class.
IEnumerator<T>	Provides members that enable the contents of a collection to be obtained one at a time.
IEqualityComparer<T>	Compares two objects for equality.
IList<T>	Defines a generic collection that can be accessed via an indexer.

TABLE 24-10 The Generic Collection Interfaces

ICollection<T> defines the following methods. Notice it defines a few more methods than does its non-generic counterpart.

Method	Description
void Add(T *obj*)	Adds *obj* to the invoking collection. Throws a **NotSupportedException** if the collection is read-only.
void Clear( )	Deletes all elements from the invoking collection and sets **Count** to zero.
bool Contains(T *obj*)	Returns true if the invoking collection contains the object passed in *obj* and false otherwise.
void CopyTo(T[ ] *target*, int *startIdx*)	Copies the contents of the invoking collection to the array specified by *target*, beginning at the index specified by *startIdx*.
bool Remove(T *obj*)	Removes the first occurrence of *obj* from the invoking collection. Returns true if *obj* was removed and false if it was not found in the invoking collection.

Several of these methods will throw **NotSupportedException** if the collection is read-only. Because **ICollection<T>** inherits **IEnumerable** and **IEnumerable<T>**, it also includes both the generic and non-generic forms of the method **GetEnumerator( )**.

Because **ICollection<T>** inherits **IEnumerable<T>**, it supports the extension methods defined by **Enumerable**. Although the extension methods were designed mostly for LINQ, they are available for other uses, including collections.

The IList<T> Interface

The **IList<T>** interface defines the behavior of a generic collection that allows elements to be accessed via a zero-based index. It inherits **IEnumerable**, **IEnumerable<T>**, and **ICollection<T>** and is the generic version of the non-generic **IList** interface. **IList<T>** defines the methods shown in Table 24-11. Two of these methods imply the modification of a collection. If the collection is read-only or of fixed size, then the **Insert( )** and **RemoveAt( )** methods will throw a **NotSupportedException**.

IList<T> defines the following indexer:

```
T this[int idx] { get; set; }
```

This indexer sets or gets the value of the element at the index specified by *idx*.

Method	Description
int IndexOf(T *obj*)	Returns the index of the first occurrence of *obj* if *obj* is contained within the invoking collection. If *obj* is not found, –1 is returned.
void Insert(int *idx*, T *obj*)	Inserts *obj* at the index specified by *idx*.
void RemoveAt(int *idx*)	Removes the object at the index specified by *idx* from the invoking collection.

TABLE 24-11 The Methods Defined by **IList<T>**

The IDictionary<TK, TV> Interface

The **IDictionary<TK, TV>** interface defines the behavior of a generic collection that maps unique keys to values. That is, it defines a collection that stores key/value pairs. **IDictionary<TK, TV>** inherits **IEnumerable**, **IEnumerable<KeyValuePair<TK, TV>>**, and **ICollection< KeyValuePair<TK, TV>>** and is the generic version of the non-generic **IDictionary**. The methods declared by **IDictionary<TK, TV>** are summarized in Table 24-12. All throw an **ArgumentNullException** if an attempt is made to specify a null key.

IDictionary<TK, TV> defines the following properties:

Property	Description
ICollection Keys<TK> { get; }	Obtains a collection of the keys.
ICollection Values<TV> { get; }	Obtains a collection of the values.

Notice that the keys and values contained within the collection are available as separate lists through the **Keys** and **Values** properties.

IDictionary<TK, TV> defines the following indexer:

```
TV this[TK key] { get; set; }
```

You can use this indexer to get or set the value of an element. You can also use it to add a new element to the collection. Notice that the "index" is not actually an index, but rather the key of the item.

IEnumerable<T> and IEnumerator<T>

IEnumerable<T> and **IEnumerator<T>** are the generic equivalents of the non-generic **IEnumerable** and **IEnumerator** interfaces described earlier. They declare the same methods and properties, and work in the same way. Of course, the generic versions operate on data of the type specified by the type argument.

IEnumerable<T> declares the **GetEnumerator( )** method as shown here:

```
IEnumerator<T> GetEnumerator( )
```

It returns an enumerator of type **T** for the collection. Thus, it returns a type-safe enumerator.

Method	Description
void Add(TK *k*, TV *v*)	Adds the key/value pair specified by *k* and *v* to the invoking collection. An **ArgumentException** is thrown if *k* is already stored in the collection.
bool ContainsKey(TK *k*)	Returns true if the invoking collection contains *k* as a key. Otherwise, returns false.
bool Remove(TK *k*)	Removes the entry whose key equals *k*.
bool TryGetValue(TK *k*, out TV *v*)	Attempts to retrieve the value associated with *k*, putting it into *v*. Returns true if successful and false otherwise. If *k* is not found, *v* is given its default value.

TABLE 24-12 The Methods Defined by **IDictionary<TK, TV>**

IEnumerator<T> has the same two methods as does the non-generic **IEnumerator**: **MoveNext( )** and **Reset( )**. It also declares a generic version of the **Current** property, as shown here:

```
T Current { get; }
```

It returns a **T** reference to the next object. Thus, the generic version of **Current** is type-safe.

There is one other difference between **IEnumerator** and **IEnumerator<T>**: **IEnumerator<T>** inherits the **IDisposable** interface, but **IEnumerator** does not. **IDisposable** defines the **Dispose( )** method, which is used to free unmanaged resources.

NOTE *IEnumerable<T> also implements the non-generic **IEnumerable** interface. Thus, it supports the non-generic version of **GetEnumerator( )**. **IEnumerator<T>** also implements the non-generic **IEnumerator** interface, thus supporting the non-generic versions of **Current**.*

IComparer<T>

The **IComparer<T>** interface is the generic version of **IComparer** described earlier. The main difference between the two is that **IComparer<T>** is type-safe, declaring the generic version of **Compare( )** shown here:

```
int Compare(T obj1, T obj2)
```

This method compares *obj1* with *obj2* and returns greater than zero if *obj1* is greater than *obj2*, zero if the two objects are the same, and less than zero if *obj1* is less that *obj2*.

IEqualityComparer<T>

The **IEqualityComparer<T>** interface is the equivalent of its non-generic relative **IEqualityComparer**. It defines these two methods:

```
bool Equals(T obj1, T obj2)

int GetHashCode(T obj)
```

Equals( ) must return true if **obj1** and **obj2** are equal. **GetHashCode( )** must return the hash code for *obj*.

The KeyValuePair<TK, TV> Structure

System.Collections.Generic defines a structure called **KeyValuePair<TK, TV>**, which is used to store a key and its value. It is used by the generic collection classes that store key/value pairs, such as **Dictionary<TK, TV>**. This structure defines the following two properties:

```
public TK Key { get; };
public TV Value { get; };
```

These properties hold the key or value associated with an entry. You can construct a **KeyValuePair<TK, TV>** object by using the following constructor:

```
public KeyValuePair(TK k, TV v)
```

Here, *k* is the key and *v* is the value.

The Generic Collection Classes

As mentioned at the start of this section, the generic collection classes largely parallel their non-generic relatives, although in some cases the names have been changed. Also, some differences in organization and functionality exist. The generic collections are defined in **System.Collections.Generic**. The ones described in this chapter are shown in Table 24-13. These classes form the core of the generic collections.

NOTE *System.Collections.Generic also includes the following classes:* ***SynchronizedCollection<T>*** *is a synchronized collection based on* ***IList<T>****.* ***SynchronizedReadOnlyCollection<T>*** *is a read-only synchronized collection based on* ***IList<T>****.* ***SynchronizedKeyCollection<K, V>*** *is an abstract class used as a base class by* ***System.ServiceModel.UriSchemeKeyedCollection****.* ***KeyedByTypeCollection<T>*** *is a collection that uses types as keys.*

The List<T> Collection

The **List<T>** class implements a generic dynamic array and is conceptually similar to the non-generic **ArrayList** class. **List<T>** implements the **ICollection**, **ICollection<T>**, **IList**, **IList<T>**, **IEnumerable**, and **IEnumerable<T>** interfaces. **List<T>** has the constructors shown here:

```
public List( )
public List(IEnumerable<T> c)
public List(int capacity)
```

The first constructor builds an empty **List** with a default initial capacity. The second constructor builds a **List** that is initialized with the elements of the collection specified by *c* and with an initial capacity at least equal to the number of elements. The third constructor builds an array list that has the specified initial *capacity.* The capacity is the size of the underlying array that is used to store the elements. The capacity grows automatically as elements are added to a **List<T>**. Each time the list must be enlarged, its capacity is increased.

Class	Description
Dictionary<TK, TV>	Stores key/value pairs. Provides functionality similar to that found in the non-generic **Hashtable** class.
HashSet<T>	Stores a set of unique values using a hash table.
LinkedList<T>	Stores elements in a doubly linked list.
List<T>	A dynamic array. Provides functionality similar to that found in the non-generic **ArrayList** class.
Queue<T>	A first-in, first-out list. Provides functionality similar to that found in the non-generic **Queue** class.
SortedDictionary<TK, TV>	A sorted list of key/value pairs.
SortedList<TK, TV>	A sorted list of key/value pairs. Provides functionality similar to that found in the non-generic **SortedList** class.
Stack<T>	A first-in, last-out list. Provides functionality similar to that found in the non-generic **Stack** class.

TABLE 24-13 The Core Generic Collection Classes

In addition to the methods defined by the interfaces that it implements, **List<T>** defines several methods of its own. A sampling is shown in Table 24-14.

Method	Description
public void AddRange(IEnumerable<T> *c*)	Adds the elements in *c* to the end of the invoking list.
public virtual int BinarySearch(T *v*)	Searches the invoking collection for the value passed in *v.* The index of the matching element is returned. If the value is not found, a negative value is returned. The invoking list must be sorted.
public int BinarySearch(T *v*, IComparer<T> *comp*)	Searches the invoking collection for the value passed in *v* using the comparison object specified by *comp.* The index of the matching element is returned. If the value is not found, a negative value is returned. The invoking list must be sorted.
public int BinarySearch(int *startIdx*, int *count*, T *v*, IComparer<T> *comp*)	Searches the invoking collection for the value passed in *v* using the comparison object specified by *comp.* The search begins at *startIdx* and runs for *count* elements. The index of the matching element is returned. If the value is not found, a negative value is returned. The invoking list must be sorted.
public List<T> GetRange(int *idx*, int *count*)	Returns a portion of the invoking list. The range returned begins at *idx* and runs for *count* elements. The returned object refers to the same elements as the invoking object.
public int IndexOf(T *v*)	Returns the index of the first occurrence of *v* in the invoking collection. Returns –1 if *v* is not found.
public void InsertRange(int *startIdx*, IEnumerable<T> *c*)	Inserts the elements of *c* into the invoking collection, starting at the index specified by *startIdx.*
public int LastIndexOf(T *v*)	Returns the index of the last occurrence of *v* in the invoking collection. Returns –1 if *v* is not found.
public void RemoveRange(int *idx*, int *count*)	Removes *count* elements from the invoking collection, beginning at *idx.*
public void Reverse( )	Reverses the contents of the invoking collection.
public void Reverse(int *startIdx*, int *count*)	Reverses *count* elements of the invoking collection, beginning at *startIdx.*
public void Sort( )	Sorts the collection into ascending order.
public void Sort(IComparer<T> *comp*)	Sorts the collection using the specified comparison object. If *comp* is null, the default comparer for each object is used.
public void Sort(Comparison<T> *comp*)	Sorts the collection using the specified comparison delegate.

TABLE 24-14 A Sampling of Methods Defined by **List<T>**

Method	Description
public void Sort(int *startIdx*, int *count*, IComparer<T> *comp*)	Sorts a portion of the collection using the specified comparison object. The sort begins at *startIdx* and runs for *count* elements. If *comp* is null, the default comparer for each object is used.
public T[ ] ToArray( )	Returns an array that contains copies of the elements of the invoking object.
public void TrimExcess( )	Reduces the capacity of the invoking list so that it is no more than 10 percent greater than the number of elements that it currently holds.

TABLE 24-14 A Sampling of Methods Defined by **List<T>** *(continued)*

In addition to the properties defined by the interfaces that it implements, **List<T>** adds **Capacity**, shown here:

public int Capacity { get; set; }

Capacity gets or sets the capacity of the invoking list. The capacity is the number of elements that can be held before the list must be enlarged. Because a list grows automatically, it is not necessary to set the capacity manually. However, for efficiency reasons, you might want to set the capacity when you know in advance how many elements the list will contain. This prevents the overhead associated with the allocation of more memory.

The following indexer, defined by **IList<T>**, is implemented by **List<T>**, as shown here:

public T this[int *idx*] { get; set; }

It sets or gets the value of the element at the index specified by *idx*.

Here is a program the demonstrates **List<T>**. It reworks the first **ArrayList** program shown earlier in this chapter. The only changes necessary are to substitute the name **List** for **ArrayList** and to use the generic type parameters.

```
// Demonstrate List<T>.

using System;
using System.Collections.Generic;

class GenListDemo {
  static void Main() {
    // Create a list.
    List<char> lst = new List<char>();

    Console.WriteLine("Initial number of elements: " +
                      lst.Count);

    Console.WriteLine();

    Console.WriteLine("Adding 6 elements");
    // Add elements to the list.
    lst.Add('C');
    lst.Add('A');
```

```
lst.Add('E');
lst.Add('B');
lst.Add('D');
lst.Add('F');

Console.WriteLine("Number of elements: " +
                  lst.Count);

// Display the list using array indexing.
Console.Write("Current contents: ");
for(int i=0; i < lst.Count; i++)
  Console.Write(lst[i] + " ");
Console.WriteLine("\n");

Console.WriteLine("Removing 2 elements");
// Remove elements from the list.
lst.Remove('F');
lst.Remove('A');

Console.WriteLine("Number of elements: " +
                  lst.Count);

// Use foreach loop to display the list.
Console.Write("Contents: ");
foreach(char c in lst)
  Console.Write(c + " ");
Console.WriteLine("\n");

Console.WriteLine("Adding 20 more elements");
// Add enough elements to force lst to grow.
for(int i=0; i < 20; i++)
  lst.Add((char)('a' + i));
Console.WriteLine("Current capacity: " +
                  lst.Capacity);
Console.WriteLine("Number of elements after adding 20: " +
                  lst.Count);
Console.Write("Contents: ");
foreach(char c in lst)
  Console.Write(c + " ");
Console.WriteLine("\n");

// Change contents using array indexing.
Console.WriteLine("Change first three elements");
lst[0] = 'X';
lst[1] = 'Y';
lst[2] = 'Z';

Console.Write("Contents: ");
foreach(char c in lst)
  Console.Write(c + " ");
Console.WriteLine();

// Because of generic type-safety,
```

```
    // the following line is illegal.
//    lst.Add(99); // Error, not a char!
  }
}
```

The output, shown here, is the same as that produced by the non-generic version of the program:

```
Initial number of elements: 0

Adding 6 elements
Number of elements: 6
Current contents: C A E B D F

Removing 2 elements
Number of elements: 4
Contents: C E B D

Adding 20 more elements
Current capacity: 32
Number of elements after adding 20: 24
Contents: C E B D a b c d e f g h i j k l m n o p q r s t

Change first three elements
Contents: X Y Z D a b c d e f g h i j k l m n o p q r s t
```

LinkedList<T>

The **LinkedList<T>** class implements a generic doubly linked list. It implements **ICollection**, **ICollection<T>**, **IEnumerable**, **IEnumerable<T>**, **ISerializable**, and **IDeserializationCallback**. (The last two interfaces support the serialization of the list.) **LinkedList<T>** defines two public constructors, shown here:

```
public LinkedList( )
public LinkedList(IEnumerable<T> c)
```

The first creates an empty linked list. The second creates a list initialized with the elements in *c*.

Like most linked list implementations, **LinkedList<T>** encapsulates the values stored in the list in *nodes* that contain links to the previous and next elements in the list. These nodes are objects of type **LinkedListNode<T>**. **LinkedListNode<T>** provides the four properties shown here:

```
public LinkedListNode<T> Next { get; }

public LinkedListNode<T> Previous { get; }

public LinkedList<T> List { get; }

public T Value { get; set; }
```

Next and **Previous** obtain a reference to the next or previous node in the list, respectively. You can use these properties to traverse the list in either direction. A null reference is returned if no next or previous node exists. You can obtain a reference to the list itself via **List**. You can get or set the value within a node by using **Value**.

LinkedList<T> defines many methods. A sampling is shown in Table 24-15. In addition to the properties defined by the interfaces that it implements, **LinkedList<T>** defines these properties:

```
public LinkedListNode<T> First { get; }
public LinkedListNode<T> Last { get; }
```

First obtains the first node in the list. **Last** obtains the last node in the list.

Method	Description
public LinkedListNode<T> AddAfter(LinkedListNode<T> *n*, T *v*)	Adds a node with the value *v* to the list immediately after the node specified by *n*. The node passed in *n* must not be **null**. Returns a reference to the node containing the value *v*.
public void AddAfter(LinkedListNode<T> *n*, LinkedListNode<T> *new*)	Adds the node passed in *new* to the list immediately after the node specified by *n*. The node passed in *n* must not be **null**. Throws an **InvalidOperationException** if *n* is not in the list or if *new* is part of another list.
public LinkedListNode<T> AddBefore(LinkedListNode<T> n, T v)	Adds a node with the value *v* to the list immediately before the node specified by *n*. The node passed in *n* must not be **null**. Returns a reference to the node containing the value *v*.
public void AddBefore(LinkedListNode<T> *n*, LinkedListNode<T> *new*)	Adds the node passed in *new* to the list immediately before the node specified by *n*. The node passed in *n* must not be **null**. Throws an **InvalidOperationException** if *n* is not in the list or if *new* is part of another list.
public LinkedList<T> AddFirst(T *v*)	Adds a node with the value *v* to the start of the list. Returns a reference to the node containing the value *v*.
public void AddFirst(LinkedListNode *new*)	Adds *new* to the start of the list. Throws an **InvalidOperationException** if *new* is part of another list.
public LinkedList<T> AddLast(T *v*)	Adds a node with the value *v* to the end of the list. Returns a reference to the node containing the value *v*.
public void AddLast(LinkedListNode *new*)	Adds *new* to the end of the list. Throws an **InvalidOperationException** if *new* is part of another list.
public LinkedList<T> Find(T *v*)	Returns a reference to the first node in the list that has the value *v*. **null** is returned if *v* is not in the list.
public LinkedList<T> FindLast(T *v*)	Returns a reference to the last node in the list that has the value *v*. **null** is returned if *v* is not in the list.
public bool Remove(T *v*)	Removes the first node in the list that has the value *v*. Returns true if the node was removed. (That is, if a node with the value *v* was in the list and it was removed.) Returns false otherwise.

TABLE 24-15 A Sampling of Methods Defined by **LinkedList<T>**

Method	Description
public void Remove(LinkedList<T> *n*)	Removes the node that matches *n*. Throws an **InvalidOperationException** if *n* is not in the list.
public void RemoveFirst( )	Removes the first node in the list.
public void RemoveLast( )	Removes the last node in the list.

TABLE 24-15 A Sampling of Methods Defined by **LinkedList<T>** *(continued)*

Here is an example that demonstrates the **LinkedList<T>** class:

```
// Demonstrate LinkedList<T>.

using System;
using System.Collections.Generic;

class GenLinkedListDemo {
  static void Main() {
    // Create a linked list.
    LinkedList<char> ll = new LinkedList<char>();

    Console.WriteLine("Initial number of elements: " +
                      ll.Count);

    Console.WriteLine();

    Console.WriteLine("Adding 5 elements.");
    // Add elements to the linked list
    ll.AddFirst('A');
    ll.AddFirst('B');
    ll.AddFirst('C');
    ll.AddFirst('D');
    ll.AddFirst('E');

    Console.WriteLine("Number of elements: " +
                      ll.Count);

    // Display the linked list by manually walking
    // through the list.
    LinkedListNode<char> node;

    Console.Write("Display contents by following links: ");
    for(node = ll.First; node != null; node = node.Next)
      Console.Write(node.Value + " ");

    Console.WriteLine("\n");

    //Display the linked list by use of a foreach loop.
    Console.Write("Display contents with foreach loop: ");
    foreach(char ch in ll)
      Console.Write(ch + " ");

    Console.WriteLine("\n");
```

```
    // Display the list backward by manually walking
    // from last to first.
    Console.Write("Follow links backwards: ");
      for(node = ll.Last; node != null; node = node.Previous)
      Console.Write(node.Value + " ");

    Console.WriteLine("\n");

    // Remove two elements.
    Console.WriteLine("Removing 2 elements.");
    // Remove elements from the linked list.
    ll.Remove('C');
    ll.Remove('A');

    Console.WriteLine("Number of elements: " +
                      ll.Count);

    // Use foreach loop to display the modified list.
    Console.Write("Contents after deletion: ");
    foreach(char ch in ll)
      Console.Write(ch + " ");

    Console.WriteLine("\n");

    // Add three elements to the end of the list.
    ll.AddLast('X');
    ll.AddLast('Y');
    ll.AddLast('Z');

    Console.Write("Contents after addition to end: ");
    foreach(char ch in ll)
      Console.Write(ch + " ");

    Console.WriteLine("\n");
  }
}
```

Here is the output:

```
Initial number of elements: 0

Adding 5 elements.
Number of elements: 5
Display contents by following links: E D C B A

Display contents with foreach loop: E D C B A

Follow links backwards: A B C D E

Removing 2 elements.
Number of elements: 3
Contents after deletion: E D B

Contents after addition to end: E D B X Y Z
```

Perhaps the most important thing to notice in this program is that the list is traversed in both the forward and backward direction by following the links provided by the **Next** and **Previous** properties. The bidirectional property of doubly linked lists is especially important in applications such as databases in which the ability to move efficiently through the list in both directions is often necessary.

The Dictionary<TK, TV> Class

The **Dictionary<TK, TV>** class stores key/value pairs. In a dictionary, values are accessed through their keys. In this regard, it is similar to the non-generic **Hashtable** class. **Dictionary<TK, TV>** implements **IDictionary**, **IDictionary<TV, TV>**, **ICollection**, **ICollection<KeyValuePair<TK, TV>>**, **IEnumerable**, **IEnumerable<KeyValuePair<TK, TV>>**, **ISerializable**, and **IDeserializationCallback**. (The last two interfaces support the serialization of the list.) Dictionaries are dynamic, growing as needed.

Dictionary<TK, TV> provides many constructors. Here is a sampling:

public Dictionary( )

public Dictionary(IDictionary<TK, TV> *dict*)

public Dictionary(int *capacity*)

The first constructor creates an empty dictionary with a default capacity. The second creates a dictionary that contains the same elements as those in *dict*. The third lets you specify an initial capacity. If you know in advance that you will need a dictionary of a certain size, then specifying that capacity will prevent the resizing of the dictionary at runtime, which is a costly process.

Dictionary<TK, TV> defines several methods. Some commonly used ones are shown in Table 24-16.

Method	Description
public void Add(TK *k*, TV *v*)	Adds the key/value pair specified by *k* and *v* to the dictionary. If *k* is already in the dictionary, then its value is unchanged and an **ArgumentException** is thrown.
public bool ContainsKey(TK *k*)	Returns true if *k* is a key in the invoking dictionary. Returns false otherwise.
public bool ContainsValue(TV *v*)	Returns true if *v* is a value in the invoking dictionary. Returns false otherwise.
public IDictionary.Enumerator<TK, TV> GetEnumerator( )	Returns an enumerator for the invoking dictionary.
public bool Remove(TK *k*)	Removes *k* from the dictionary. Returns true if successful. Returns false if *k* was not in the dictionary.

Table 24-16 Several Commonly Used Methods Defined by **Dictionary<TK, TV>**

In addition to the properties defined by the interfaces that it implements, **Dictionary<TK, TV>** defines these properties:

Property	Description
public IEqualityComparer<TK> Comparer { get; }	Obtains the comparer for the invoking dictionary.
public Dictionary<TK, TV>.KeyCollection Keys { get; }	Obtains a collection of the keys.
public Dictionary<TK, TV>.ValueCollection Values { get; }	Obtains a collection of the values.

Notice that the keys and values contained within the collection are available as separate lists through the **Keys** and **Values** properties. The types **Dictionary<TK, TV>.KeyCollection** and **Dictionary<TK, TV>.ValueCollection** are collections that implement both the generic and non-generic forms of **ICollection** and **IEnumerable**.

The following indexer, defined by **IDictionary<TK, TV>**, is implemented by **Dictionary<TK, TV>** as shown here:

public TV this[TK *key*] { get; set; }

You can use this indexer to get or set the value of an element. You can also use it to add a new element to the collection. Notice that the "index" is not actually an index, but rather the key of the item.

When enumerating the collection, **Dictionary<TK, TV>** returns key/value pairs in the form of a **KeyValuePair<TK, TV>** structure. Recall that this structure defines the following two fields:

public TK Key;
public TV Value;

These fields hold the key or value associated with an entry. Most of the time you won't need to use **KeyValuePair<TK, TV>** directly because **Dictionary<TK, TV>** allows you to work the keys and values individually. However, when enumerating a **Dictionary<TK, TV>**, such as in a **foreach** loop, the objects being enumerated are **KeyValuePair**s.

In a **Dictionary<TK, TV>**, all keys must be unique, and a key must not change while it is in use as a key. Values need not be unique. The objects in a **Dictionary<TK, TV>** are not stored in sorted order.

Here is an example that demonstrates **Dictionary<TK, TV>**:

```
// Demonstrate the generic Dictionary<TK, TV> class.

using System;
using System.Collections.Generic;

class GenDictionaryDemo {
```

```
  static void Main() {
    // Create a Dictionary that holds employee
    // names and their corresponding salary.
    Dictionary<string, double> dict =
      new Dictionary<string, double>();

    // Add elements to the collection.
    dict.Add("Butler, John", 73000);
    dict.Add("Swartz, Sarah", 59000);
    dict.Add("Pyke, Thomas", 45000);
    dict.Add("Frank, Ed", 99000);

    // Get a collection of the keys (names).
    ICollection<string> c = dict.Keys;

    // Use the keys to obtain the values (salaries).
    foreach(string str in c)
      Console.WriteLine("{0}, Salary: {1:C}", str, dict[str]);
  }
}
```

Here is the output:

```
Butler, John, Salary: $73,000.00
Swartz, Sarah, Salary: $59,000.00
Pyke, Thomas, Salary: $45,000.00
Frank, Ed, Salary: $99,000.00
```

The SortedDictionary<TK, TV> Class

The **SortedDictionary<TV, TK>** class stores key/value pairs and is similar to **Dictionary<TK, TV>** except that it is sorted by key. **SortedDictionary<TK, TV>** implements **IDictionary**, **IDictionary<TK, TV>**, **ICollection**, **ICollection<KeyValuePair<TK, TV>>**, **IEnumerable**, and **IEnumerable<KeyValuePair<TK, TV>>**. **SortedDictionary<TK, TV>** provides the following constructors:

public SortedDictionary( )

public SortedDictionary(IDictionary<TK, TV> *dict*)

public SortedDictionary(IComparer<TK> *comp*)

public SortedDictionary(IDictionary<TK, TV> *dict*, IComparer<TK> *comp*)

The first constructor creates an empty dictionary. The second creates a dictionary that contains the same elements as those in *dict*. The third lets you specify the **IComparer** that the dictionary will use for sorting, and the fourth lets you initialize the dictionary and specify the **IComparer**.

SortedDictionary<TK, TV> defines several methods. A sampling is shown in Table 24-17.

Method	Description
public void Add(TK *k*, TV *v*)	Adds the key/value pair specified by *k* and *v* to the dictionary. If *k* is already in the dictionary, then its value is unchanged and an **ArgumentException** is thrown.
public bool ContainsKey(TK *k*)	Returns true if *k* is a key in the invoking dictionary. Returns false otherwise.
public bool ContainsValue(TV *v*)	Returns true if *v* is a value in the invoking dictionary. Returns false otherwise.
public SortedDictionary.Enumerator<TK, TV> GetEnumerator( )	Returns an enumerator for the invoking dictionary.
public bool Remove(TK *k*)	Removes *k* from the dictionary. Returns true if successful. Returns false if *k* was not in the dictionary.

TABLE 24-17 A Sampling of Methods Defined by **SortedDictionary<TK, TV>**

In addition to the properties defined by the interfaces that it implements, **SortedDictionary<TK, TV>** defines the following properties:

Property	Description
public IComparer<TK> Comparer { get; }	Obtains the comparer for the invoking dictionary.
public SortedDictionary<TK, TV>.KeyCollection Keys { get; }	Obtains a collection of the keys.
public SortedDictionary<TK, TV>.ValueCollection Values { get; }	Obtains a collection of the values.

Notice that the keys and values contained within the collection are available as separate lists through the **Keys** and **Values** properties. The types

SortedDictionary<TK, TV>.KeyCollection
SortedDictionary<TK, TV>.ValueCollection

are collections that implement both the generic and non-generic forms of **ICollection** and **IEnumerable**.

SortedDictionary<TK, TV> defines the following indexer (which is specified by **IDictionary<TK, TV>**):

public TV this[TK *key*] { get; set; }

You can use this indexer to get or set the value of an element. You can also use it to add a new element to the collection. Notice that the "index" is not actually an index, but rather the key of the item.

When enumerated, **SortedDictionary<TK, TV>** returns key/value pairs in the form of a **KeyValuePair<TK, TV>** structure. Recall that this structure defines the following two fields:

```
public TK Key;
public TV Value;
```

These fields hold the key or value associated with an entry. Most of the time you won't need to use **KeyValuePair<TK, TV>** directly because **SortedDictionary<TK, TV>** allows you to work the keys and values individually. However, when enumerating a **SortedDictionary<TK, TV>**, such as in a **foreach** loop, the objects being enumerated are **KeyValuePair**s.

In a **SortedDictionary<TK, TV>**, all keys must be unique, and a key must not change while it is in use as a key. Values need not be unique.

Here is an example that demonstrates **SortedDictionary<TK, TV>**. It reworks the **Dictionary<TK, TV>** example shown in the preceding section. In this version, the database of employees and salaries is sorted based on name (which is the key).

```
// Demonstrate the generic SortedDictionary<TK, TV> class.

using System;
using System.Collections.Generic;

class GenSortedDictionaryDemo {
  static void Main() {
    // Create a Dictionary that holds employee
    // names and their corresponding salary.
    SortedDictionary<string, double> dict =
      new SortedDictionary<string, double>();

    // Add elements to the collection.
    dict.Add("Butler, John", 73000);
    dict.Add("Swartz, Sarah", 59000);
    dict.Add("Pyke, Thomas", 45000);
    dict.Add("Frank, Ed", 99000);

    // Get a collection of the keys (names).
    ICollection<string> c = dict.Keys;

    // Use the keys to obtain the values (salaries).
    foreach(string str in c)
      Console.WriteLine("{0}, Salary: {1:C}", str, dict[str]);
  }
}
```

The output is shown here:

```
Butler, John, Salary: $73,000.00
Frank, Ed, Salary: $99,000.00
Pyke, Thomas, Salary: $45,000.00
Swartz, Sarah, Salary: $59,000.00
```

As you can see, the list is now sorted based on the key, which is the employee's name.

The SortedList<TK, TV> Class

The **SortedList<TK, TV>** class stores a sorted list of key/value pairs. It is the generic equivalent of the non-generic **SortedList** class. **SortedList<TK, TV>** implements **IDictionary**, **IDictionary<TK, TV>**, **ICollection**, **ICollection< KeyValuePair<TK, TV>>**, **IEnumerable**, and **IEnumerable< KeyValuePair<TK, TV>>**. The size of a **SortedList<TK, TV>** is dynamic and will automatically grow as needed. **SortedList<TV, TK>** is similar to **SortedDictionary<TK, TV>** but has different performance characteristics. For example, a **SortedList<TK, TV>** uses less memory, but a **SortedDictionary<TK, TV>** is faster when inserting out-of-order elements.

SortedList<TK, TV> provides many constructors. Here is a sampling:

public SortedList( )

public SortedList(IDictionary<TK, TV> *dict*)

public SortedList(int *capacity*)

public SortedList(IComparer<TK> *comp*)

The first constructor creates an empty list with a default capacity. The second creates a list that contains the same elements as those in *dict*. The third lets you specify an initial capacity. If you know in advance that you will need a list of a certain size, then specifying that capacity will prevent the resizing of the list at runtime, which is a costly process. The fourth form lets you specify a comparison method that will be used to compare the objects contained in the list.

The capacity of a **SortedList<TK, TV>** list grows automatically as needed when elements are added to a list. When the current capacity is exceeded, the capacity is increased. The advantage of specifying a capacity is that you can prevent or minimize the overhead associated with resizing the collection. Of course, it makes sense to specify an initial capacity only if you have some idea of how many elements will be stored.

In addition to the methods defined by the interfaces that it implements, **SortedList<TK, TV>** also defines several methods of its own. A sampling is shown in Table 24-18. Notice that the enumerator returned by **GetEnumerator( )** enumerates the key/value pairs stored in the list as objects of type **KeyValuePair**.

Method	Description
public void Add(TK *k*, TV *v*)	Adds the key/value pair specified by *k* and *v* to the list. If *k* is already in the list, then its value is unchanged and an **ArgumentException** is thrown.
public bool ContainsKey(TK *k*)	Returns true if *k* is a key in the invoking list. Returns false otherwise.
public bool ContainsValue(TV *v*)	Returns true if *v* is a value in the invoking list. Returns false otherwise.
public IEnumerator<KeyValuePair<TK, TV>> GetEnumerator( )	Returns an enumerator for the invoking list.
public int IndexOfKey(TK *k*)	Returns the index of the key specified by *k*. Returns –1 if the key is not in the list.

TABLE 24-18 Several Commonly Used Methods Defined by **SortedList<TK, TV>**

Method	Description
public int IndexOfValue(TV *v*)	Returns the index of the first occurrence of the value specified by *v*. Returns –1 if the value is not in the list.
public bool Remove(TK *k*)	Removes the key/value pair associated with *k* from the list. Returns true if successful. Returns false if *k* is not in the list.
public void RemoveAt(int *idx*)	Removes the key/value pair at the index specified by *idx*.
public void TrimExcess( )	Removes the excess capacity of the invoking list.

TABLE 24-18 Several Commonly Used Methods Defined by **SortedList<TK, TV>** *(continued)*

In addition to the properties defined by the interfaces that it implements, **SortedList<TK, TV>** defines the following properties:

Property	Description
public int Capacity { get; set; }	Obtains or sets the capacity of the invoking list.
public IComparer<TK> Comparer { get; }	Obtains the comparer for the invoking list.
public IList<TK> Keys { get; }	Obtains a collection of the keys.
public IList<TV> Values { get; }	Obtains a collection of the values.

SortedList<TK, TV> defines the following indexer (which is defined by **IDictionary<TK, TV>**):

public TV this[TK *key*] { get; set; }

You can use this indexer to get or set the value of an element. You can also use it to add a new element to the collection. Notice that the "index" is not actually an index, but rather the key of the item.

Here is an example that demonstrates **SortedList<TK, TV>**. It reworks the employee database example one more time. In this version, the database is stored in a **SortedList**.

```
// Demonstrate a SortedList<TK, TV>.

using System;
using System.Collections.Generic;

class GenSLDemo {
  static void Main() {
    // Create a sorted SortedList for
    // employee names and salary.
    SortedList<string, double> sl =
      new SortedList<string, double>();
```

```
    // Add elements to the collection.
    sl.Add("Butler, John", 73000);
    sl.Add("Swartz, Sarah", 59000);
    sl.Add("Pyke, Thomas", 45000);
    sl.Add("Frank, Ed", 99000);

    // Get a collection of the keys.
    ICollection<string> c = sl.Keys;

    // Use the keys to obtain the values.
    foreach(string str in c)
      Console.WriteLine("{0}, Salary: {1:C}", str, sl[str]);

    Console.WriteLine();
  }
}
```

The output is shown here:

```
Butler, John, Salary: $73,000.00
Frank, Ed, Salary: $99,000.00
Pyke, Thomas, Salary: $45,000.00
Swartz, Sarah, Salary: $59,000.00
```

As the output shows, the list is sorted based on employee name, which is the key.

The Stack<T> Class

Stack<T> is the generic equivalent of the non-generic **Stack** class. **Stack<T>** supports a first-in, last-out stack. It implements the **ICollection**, **IEnumerable**, and **IEnumerable<T>** interfaces. **Stack<T>** directly implements the **Clear( )**, **Contains( )**, and **CopyTo( )** methods defined by **ICollection<T>**. (The **Add( )** and **Remove( )** methods are not supported, nor is the **IsReadOnly** property.) **Stack<T>** is a dynamic collection that grows as needed to accommodate the elements it must store. It defines the following constructors:

public Stack( )
public Stack(int *capacity*)
public Stack(IEnumerable<T> *c*)

The first form creates an empty stack with a default initial capacity. The second form creates an empty stack with the initial capacity specified by *capacity.* The third form creates a stack that contains the elements of the collection specified by *c.*

In addition to the methods defined by the interfaces that it implements (and those methods defined by **ICollection<T>** that it implements on its own), **Stack<T>** defines the methods shown in Table 24-19. **Stack<T>** works just like its non-generic counterpart. To put an object on the top of the stack, call **Push( )**. To remove and return the top element, call **Pop( )**. You can use **Peek( )** to return, but not remove, the top object. An **InvalidOperationException** is thrown if you call **Pop( )** or **Peek( )** when the invoking stack is empty.

Method	Description
public T Peek( )	Returns the element on the top of the stack, but does not remove it.
public T Pop( )	Returns the element on the top of the stack, removing it in the process.
public void Push(T *v*)	Pushes *v* onto the stack.
public T[ ] ToArray( )	Returns an array that contains copies of the elements of the invoking stack.
public void TrimExcess( )	Removes the excess capacity of the invoking stack.

TABLE 24-19 The Methods Defined by **Stack<T>**

The following program demonstrates **Stack<T>**:

```
// Demonstrate the Stack<T> class.

using System;
using System.Collections.Generic;

class GenStackDemo {
  static void Main() {
    Stack<string> st = new Stack<string>();

    st.Push("One");
    st.Push("Two");
    st.Push("Three");
    st.Push("Four");
    st.Push("Five");

    while(st.Count > 0) {
      string str = st.Pop();
      Console.Write(str + " ");
    }

    Console.WriteLine();
  }
}
```

The output is shown here:

```
Five Four Three Two One
```

The Queue<T> Class

Queue<T> is the generic equivalent of the non-generic **Queue** class. It supports a first-in, first-out list. **Queue<T>** implements the **ICollection**, **IEnumerable**, and **IEnumerable<T>** interfaces. **Queue<T>** directly implements the **Clear( )**, **Contains( )**, and **CopyTo( )** methods

defined by **ICollection<T>**. (The **Add( )** and **Remove( )** methods are not supported, nor is the **IsReadOnly** property.) **Queue<T>** is a dynamic collection that grows as needed to accommodate the elements it must store. It defines the following constructors:

public Queue( )
public Queue(int *capacity*)
public Queue(IEnumerable<T> *c*)

The first form creates an empty queue with an initial default capacity. The second form creates an empty queue with the initial capacity specified by *capacity*. The third form creates a queue that contains the elements of the collection specified by *c*.

In addition to the methods defined by the interfaces that it implements (and those methods defined by **ICollection<T>** that it implements on its own), **Queue<T>** defines the methods shown in Table 24-20. **Queue<T>** works just like its non-generic counterpart. To put an object in the queue, call **Enqueue( )**. To remove and return the object at the front of the queue, call **Dequeue( )**. You can use **Peek( )** to return, but not remove, the next object. An **InvalidOperationException** is thrown if you call **Dequeue( )** or **Peek( )** when the invoking queue is empty.

Here is an example that demonstrates **Queue<T>**:

```
// Demonstrate the Queue<T> class.

using System;
using System.Collections.Generic;

class GenQueueDemo {
  static void Main() {
    Queue<double> q = new Queue<double>();

    q.Enqueue(98.6);
    q.Enqueue(212.0);
    q.Enqueue(32.0);
    q.Enqueue(3.1416);

    double sum = 0.0;
    Console.Write("Queue contents: ");
    while(q.Count > 0) {
      double val = q.Dequeue();
      Console.Write(val + " ");
      sum += val;
    }

    Console.WriteLine("\nTotal is " + sum);
  }
}
```

The output is shown here:

```
Queue contents: 98.6 212 32 3.1416
Total is 345.7416
```

Method	Description
public T Dequeue( )	Returns the object at the front of the invoking queue. The object is removed in the process.
public void Enqueue(T *v*)	Adds *v* to the end of the queue.
public T Peek( )	Returns the object at the front of the invoking queue, but does not remove it.
public T[ ] ToArray( )	Returns an array that contains copies of the elements of the invoking queue.
public void TrimExcess( )	Removes the excess capacity of the invoking stack.

TABLE 24-20 The Methods Defined by **Queue<T>**

HashSet<T>

HashSet<T> is a new collection added to the .NET Framework by version 3.5. It supports a collection that implements a set. It uses a hash table for storage. **HashSet<T>** implements the **ICollection<T>**, **IEnumerable**, **IEnumerable<T>**, **ISerializable**, and **IDeserializationCallback** interfaces. **HashSet<T>** implements a set in which all elements are unique. In other words, duplicates are not allowed. The order of the elements is not specified. **HashSet<T>** defines a full complement of set operations, such as intersection, union, and symmetric difference. This makes **HashSet<T>** the perfect choice for working with sets of objects. **HashSet<T>** is a dynamic collection that grows as needed to accommodate the elements it must store.

Here are four commonly used constructors defined by **HashSet<T>**:

```
public HashSet( )
public HashSet(IEnumerable<T> c)
public HashSet(IEqualityCompare comp)
public HashSet(IEnumerable<T> c, IEqualityCompare comp)
```

The first form creates an empty set. The second creates a set that contains the elements of the collection specified by *c*. The third lets you specify the comparer. The fourth creates a set that contains the elements in the collection specified by *c* and uses the comparer specified by *comp*. There is also a fifth constructor that lets you initialize a set from serialized data.

In addition to the methods defined by the interfaces that it implements, **HashSet<T>** defines several of its own, most of which support various set operations. The set operation methods defined by **HashSet<T>** are shown in Table 24-21. Notice that the arguments to these methods are **IEnumerable<T>**. This means you can pass something other than another **HashSet<T>** as the second set. Most often, however, both operands will be instances of **HashSet<T>**.

In addition to the properties defined by **ICollection<T>**, **HashSet<T>** adds **Comparer**, shown here:

```
public IEqualityComparer<T> Comparer { get; }
```

It obtains the comparer for the invoking hash set.

Method	Description
public void ExceptWith(IEnumerable<T> *set2*)	Removes the elements in *set2* from the invoking set.
public void IntersectWith(IEnumerable<T> *set2*)	Removes from the invoking set those elements not common to both the invoking set and *set2.*
public bool IsProperSubsetOf(IEnumerable<T> *set2*)	Returns true if the invoking set is a proper subset of *set2.*
public bool IsProperSupersetOf(IEnumerable<T> *set2*)	Returns true if the invoking set is a proper superset of *set2.*
public bool IsSubsetOf(IEnumerable<T> *set2*)	Returns true if the invoking set is a subset of *set2.*
public bool IsSuperSetOf(IEnumerable<T> *set2*)	Returns true if the invoking set is a superset of *set2.*
public bool SetEquals(IEnumerable<T> *set2*)	Returns true if the invoking set is equivalent to *set2.* This determination is independent of the order of the elements.
public void SymmetricExceptWith(IEnumerable<T> *set2*)	Changes the invoking set so that it contains all elements from both the invoking set and *set2,* except for those elements common to both sets. This is usually called the symmetric difference of the two sets.
public void UnionWith(IEnumerable<T> *set2*)	Adds the elements from *set2* to the invoking set. Duplicates are not included. Thus, it creates a union of the two sets.

TABLE 24-21 The Set Operations Defined by **HashSet<T>**

Here is an example that shows **HashSet<T>** in action:

```
// Demonstrate the HashSet<T> class.

using System;
using System.Collections.Generic;

class HashSetDemo {

  static void Show(string msg, HashSet<char> set) {
    Console.Write(msg);
    foreach(char ch in set)
      Console.Write(ch + " ");
    Console.WriteLine();
  }

  static void Main() {
    HashSet<char> setA = new HashSet<char>();
    HashSet<char> setB = new HashSet<char>();

    setA.Add('A');
```

```
    setA.Add('B');
    setA.Add('C');

    setB.Add('C');
    setB.Add('D');
    setB.Add('E');

    Show("Initial content of setA: ", setA);
    Show("Initial content of setB: ", setB);

    setA.SymmetricExceptWith(setB);
    Show("setA after Symmetric difference with SetB: ", setA);

    setA.UnionWith(setB);
    Show("setA after union with setB: ", setA);

    setA.ExceptWith(setB);
    Show("setA after subtracting setB: ", setA);

    Console.WriteLine();
  }
}
```

The output is shown here:

```
Initial content of setA: A B C
Initial content of setB: C D E
setA after Symmetric difference with SetB: A B D E
setA after union with setB: A B D E C
setA after subtracting setB: A B
```

Storing User-Defined Classes in Collections

For the sake of simplicity, the foregoing examples have stored built-in types, such as **int**, **string**, or **char**, in a collection. Of course, collections are not limited to the storage of built-in objects. Quite the contrary. The power of collections is that they can store any type of object, including objects of classes that you create.

Let's begin with an example that uses the non-generic class **ArrayList** to store inventory information that is encapsulated by the **Inventory** class:

```
// A simple inventory example.

using System;
using System.Collections;

class Inventory {
  string name;
  double cost;
  int onhand;

  public Inventory(string n, double c, int h) {
    name = n;
    cost = c;
    onhand = h;
```

```
  }

  public override string ToString() {
    return
      String.Format("{0,-10}Cost: {1,6:C}  On hand: {2}",
                    name, cost, onhand);
  }
}

class InventoryList {
  static void Main() {
    ArrayList inv = new ArrayList();

    // Add elements to the list
    inv.Add(new Inventory("Pliers", 5.95, 3));
    inv.Add(new Inventory("Wrenches", 8.29, 2));
    inv.Add(new Inventory("Hammers", 3.50, 4));
    inv.Add(new Inventory("Drills", 19.88, 8));

    Console.WriteLine("Inventory list:");
    foreach(Inventory i in inv) {
      Console.WriteLine("   " + i);
    }
  }
}
```

The output from the program is shown here:

```
Inventory list:
   Pliers    Cost:  $5.95  On hand: 3
   Wrenches  Cost:  $8.29  On hand: 2
   Hammers   Cost:  $3.50  On hand: 4
   Drills    Cost: $19.88  On hand: 8
```

In the program, notice that no special actions were required to store objects of type **Inventory** in a collection. Because all types inherit **object**, any type of object can be stored in any non-generic collection. Thus, using a non-generic collection, it is trivially easy to store objects of classes that you create. Of course, it also means the collection is not type-safe.

To store objects of classes that you create in a type-safe collection, you must use one of the generic collection classes. For example, here is a version of the preceding program rewritten to use **List<T>**. The output is the same as before.

```
// Store Inventory Objects in a List<T> collection.

using System;
using System.Collections.Generic;

class Inventory {
  string name;
  double cost;
  int onhand;

  public Inventory(string n, double c, int h) {
    name = n;
```

```
    cost = c;
    onhand = h;
  }

  public override string ToString() {
    return
      String.Format("{0,-10}Cost: {1,6:C}  On hand: {2}",
                    name, cost, onhand);
  }
}

class TypeSafeInventoryList {
  static void Main() {
    List<Inventory> inv = new List<Inventory>();

    // Add elements to the list.
    inv.Add(new Inventory("Pliers", 5.95, 3));
    inv.Add(new Inventory("Wrenches", 8.29, 2));
    inv.Add(new Inventory("Hammers", 3.50, 4));
    inv.Add(new Inventory("Drills", 19.88, 8));

    Console.WriteLine("Inventory list:");
    foreach(Inventory i in inv) {
      Console.WriteLine("   " + i);
    }
  }
}
```

In this version, notice the only real difference is the passing of the type **Inventory** as a type argument to **List<T>**. Other than that, the two programs are nearly identical. The fact that the use of a generic collection requires virtually no additional effort and adds type safety argues strongly for its use when storing a specific type of object within a collection.

In general, there is one other thing to notice about the preceding programs: Both are quite short. When you consider that each sets up a dynamic array that can store, retrieve, and process inventory information in less than 40 lines of code, the power of collections begins to become apparent. As most readers will know, if all of this functionality had to be coded by hand, the program would have been several times longer. Collections offer ready-to-use solutions to a wide variety of programming problems. You should use them whenever the situation warrants.

There is one limitation to the preceding programs that may not be immediately apparent: The collection can't be sorted. The reason for this is that neither **ArrayList** nor **List<T>** has a way to compare two **Inventory** objects. There are two ways to remedy this situation. First, **Inventory** can implement the **IComparable** interface. This interface defines how two objects of a class are compared. Second, an **IComparer** object can be specified when comparisons are required. The following sections illustrate both approaches.

Implementing IComparable

If you want to sort a collection that contains user-defined objects (or if you want to store those objects in a collection such as **SortedList**, which maintains its elements in sorted order), then the collection must know how to compare those objects. One way to do this is

for the object being stored to implement the **IComparable** interface. The **IComparable** interface comes in two forms: generic and non-generic. Although the way each is used is similar, there are some small differences. Each is examined here.

Implementing IComparable for Non-Generic Collections

If you want to sort objects that are stored in a non-generic collection, then you will implement the non-generic version of **IComparable**. This version defines only one method, **CompareTo( )**, which determines how comparisons are performed. The general form of **CompareTo( )** is shown here:

int CompareTo(object *obj)*

CompareTo( ) compares the invoking object to *obj*. To sort in ascending order, your implementation must return zero if the objects are equal, a positive value if the invoking object is greater than *obj*, and a negative value if the invoking object is less than *obj*. You can sort in descending order by reversing the outcome of the comparison. The method can throw an **ArgumentException** if the type of *obj* is not compatible for comparison with the invoking object.

Here is an example that shows how to implement **IComparable**. It adds **IComparable** to the **Inventory** class developed in the preceding section. It implements **CompareTo( )** so that it compares the **name** field, thus enabling the inventory to be sorted by name. By implementing **IComparable**, it allows a collection of **Inventory** objects to be sorted, as the program illustrates.

```
// Implement IComparable.

using System;
using System.Collections;

// Implement the non-generic IComparable interface.
class Inventory : IComparable {
  string name;
  double cost;
  int onhand;

  public Inventory(string n, double c, int h) {
    name = n;
    cost = c;
    onhand = h;
  }

  public override string ToString() {
    return
      String.Format("{0,-10}Cost: {1,6:C}  On hand: {2}",
                    name, cost, onhand);
  }

  // Implement the IComparable interface.
  public int CompareTo(object obj) {
    Inventory b;
    b = (Inventory) obj;
```

```
    return name.CompareTo(b.name);
  }
}

class IComparableDemo {
  static void Main() {
    ArrayList inv = new ArrayList();

    // Add elements to the list.
    inv.Add(new Inventory("Pliers", 5.95, 3));
    inv.Add(new Inventory("Wrenches", 8.29, 2));
    inv.Add(new Inventory("Hammers", 3.50, 4));
    inv.Add(new Inventory("Drills", 19.88, 8));

    Console.WriteLine("Inventory list before sorting:");
    foreach(Inventory i in inv) {
      Console.WriteLine("   " + i);
    }
    Console.WriteLine();

    // Sort the list.
    inv.Sort();

    Console.WriteLine("Inventory list after sorting:");
    foreach(Inventory i in inv) {
      Console.WriteLine("   " + i);
    }
  }
}
```

Here is the output. Notice that after the call to **Sort( )**, the inventory is sorted by name.

```
Inventory list before sorting:
   Pliers    Cost:  $5.95  On hand: 3
   Wrenches  Cost:  $8.29  On hand: 2
   Hammers   Cost:  $3.50  On hand: 4
   Drills    Cost: $19.88  On hand: 8

Inventory list after sorting:
   Drills    Cost: $19.88  On hand: 8
   Hammers   Cost:  $3.50  On hand: 4
   Pliers    Cost:  $5.95  On hand: 3
   Wrenches  Cost:  $8.29  On hand: 2
```

Implementing IComparable<T> for Generic Collections

If you want to sort objects that are stored in a generic collection, then you will implement **IComparable<T>**. This version defines the generic form of **CompareTo( )** shown here:

int CompareTo(T *obj)*

CompareTo( ) compares the invoking object to *obj.* To sort in ascending order, your implementation must return zero if the objects are equal, a positive value if the invoking object is greater than *obj,* and a negative value if the invoking object is less than *obj.* To

sort in descending order, reverse the outcome of the comparison. When implementing **IComparable<T>**, you will usually pass the type name of the implementing class as a type argument.

The following example reworks the preceding program so that it uses **IComparable<T>**. Notice it uses the generic **List<T>** collection rather than the non-generic **ArrayList**.

```
// Implement IComparable<T>.

using System;
using System.Collections.Generic;

// Implement the generic IComparable<T> interface.
class Inventory : IComparable<Inventory> {
  string name;
  double cost;
  int onhand;

  public Inventory(string n, double c, int h) {
    name = n;
    cost = c;
    onhand = h;
  }

  public override string ToString() {
    return
      String.Format("{0,-10}Cost: {1,6:C}  On hand: {2}",
                    name, cost, onhand);
  }

  // Implement the IComparable<T> interface.
  public int CompareTo(Inventory obj) {
    return name.CompareTo(obj.name);
  }
}

class GenericIComparableDemo {
  static void Main() {
    List<Inventory> inv = new List<Inventory>();

    // Add elements to the list.
    inv.Add(new Inventory("Pliers", 5.95, 3));
    inv.Add(new Inventory("Wrenches", 8.29, 2));
    inv.Add(new Inventory("Hammers", 3.50, 4));
    inv.Add(new Inventory("Drills", 19.88, 8));

    Console.WriteLine("Inventory list before sorting:");
    foreach(Inventory i in inv) {
      Console.WriteLine("   " + i);
    }
    Console.WriteLine();

    // Sort the list.
    inv.Sort();
```

```
    Console.WriteLine("Inventory list after sorting:");
    foreach(Inventory i in inv) {
      Console.WriteLine("   " + i);
    }
  }
}
```

This program produces the same output as the previous, non-generic version.

Using an IComparer

Although implementing **IComparable** for classes that you create is often the easiest way to allow objects of those classes to be sorted, you can approach the problem in a different way by using **IComparer**. To use **IComparer**, first create a class that implements **IComparer**, and then specify an object of that class when comparisons are required.

There are two versions of **IComparer**: generic and non-generic. Although the way each is used is similar, there are some small differences, and each approach is examined here.

Using a Non-Generic IComparer

The non-generic **IComparer** defines only one method, **Compare( )**, which is shown here:

int Compare(object *obj1*, object *obj2*)

Compare( ) compares *obj1* to *obj2*. To sort in ascending order, your implementation must return zero if the objects are equal, a positive value if *obj1* is greater than *obj2*, and a negative value if *obj1* is less than *obj2*. You can sort in descending order by reversing the outcome of the comparison. The method can throw an **ArgumentException** if the type of *obj* is not compatible for comparison with the invoking object.

An **IComparer** can be specified when constructing a **SortedList**, when calling **ArrayList.Sort(IComparer)**, and at various other places throughout the collection classes. The main advantage of using **IComparer** is that you can sort objects of classes that do not implement **IComparable**.

The following program reworks the non-generic inventory program so that it uses an **IComparer** to sort the inventory list. It first creates a class called **CompInv** that implements **IComparer** and compares two **Inventory** objects. An object of this class is then used in a call to **Sort( )** to sort the inventory list.

```
// Use IComparer.

using System;
using System.Collections;

// Create an IComparer for Inventory objects.
class CompInv : IComparer {
  // Implement the IComparer interface.
  public int Compare(object obj1, object obj2) {
    Inventory a, b;
    a = (Inventory) obj1;
    b = (Inventory) obj2;
    return a.name.CompareTo(b.name);
  }
}
```

```
class Inventory {
  public string name;
  double cost;
  int onhand;

  public Inventory(string n, double c, int h) {
    name = n;
    cost = c;
    onhand = h;
  }

  public override string ToString() {
    return
      String.Format("{0,-10}Cost: {1,6:C}  On hand: {2}",
                    name, cost, onhand);
  }
}

class IComparerDemo {
  static void Main() {
    CompInv comp = new CompInv();
    ArrayList inv = new ArrayList();

    // Add elements to the list.
    inv.Add(new Inventory("Pliers", 5.95, 3));
    inv.Add(new Inventory("Wrenches", 8.29, 2));
    inv.Add(new Inventory("Hammers", 3.50, 4));
    inv.Add(new Inventory("Drills", 19.88, 8));

    Console.WriteLine("Inventory list before sorting:");
    foreach(Inventory i in inv) {
      Console.WriteLine("   " + i);
    }
    Console.WriteLine();

    // Sort the list using an IComparer.
    inv.Sort(comp);

    Console.WriteLine("Inventory list after sorting:");
    foreach(Inventory i in inv) {
      Console.WriteLine("   " + i);
    }
  }
}
```

The output is the same as the previous version of the program.

Using a Generic IComparer<T>

The **IComparer<T>** interface is the generic version of **IComparer**. It defines the generic version of **Compare( )**, shown here:

int Compare(T *obj1*, T *obj2*)

This method compares *obj1* with *obj2* and returns greater than zero if *obj1* is greater than *obj2*, zero if the two objects are the same, and less than zero if *obj1* is less that *obj2*.

Here is a generic version of the preceding program that uses **IComparer<T>**. It produces the same output as the previous versions of the program.

```
// Use IComparer<T>.

using System;
using System.Collections.Generic;

// Create an IComparer<T> for Inventory objects.
class CompInv<T> : IComparer<T> where T : Inventory {

  // Implement the IComparer<T> interface.
  public int Compare(T obj1, T obj2) {
    return obj1.name.CompareTo(obj2.name);
  }
}

class Inventory {
  public string name;
  double cost;
  int onhand;

  public Inventory(string n, double c, int h) {
    name = n;
    cost = c;
    onhand = h;
  }

  public override string ToString() {
    return
      String.Format("{0,-10}Cost: {1,6:C}  On hand: {2}",
                    name, cost, onhand);
  }
}

class GenericIComparerDemo {
  static void Main() {
    CompInv<Inventory> comp = new CompInv<Inventory>();
    List<Inventory> inv = new List<Inventory>();

    // Add elements to the list.
    inv.Add(new Inventory("Pliers", 5.95, 3));
    inv.Add(new Inventory("Wrenches", 8.29, 2));
    inv.Add(new Inventory("Hammers", 3.50, 4));
    inv.Add(new Inventory("Drills", 19.88, 8));

    Console.WriteLine("Inventory list before sorting:");
    foreach(Inventory i in inv) {
      Console.WriteLine("   " + i);
    }
    Console.WriteLine();
```

```
    // Sort the list using an IComparer.
    inv.Sort(comp);

    Console.WriteLine("Inventory list after sorting:");
    foreach(Inventory i in inv) {
      Console.WriteLine("   " + i);
    }
  }
}
```

Accessing a Collection via an Enumerator

Often you will want to cycle through the elements in a collection. For example, you might want to display each element. One way to do this is to use a **foreach** loop, as the preceding examples have done. Another way is to use an enumerator. An *enumerator* is an object that implements either the non-generic **IEnumerator** or the generic **IEnumerator<T>** interface.

IEnumerator defines one property called **Current**. The non-generic version is shown here:

```
object Current { get; }
```

For **IEnumerator<T>**, **Current** is declared like this:

```
T Current { get; }
```

In both cases, **Current** obtains the current element being enumerated. Since **Current** is a read-only property, an enumerator can only be used to retrieve, but not modify, the objects in a collection.

IEnumerator defines two methods. The first is **MoveNext( )**:

```
bool MoveNext( )
```

Each call to **MoveNext( )** moves the current position of the enumerator to the next element in the collection. It returns true if the next element is available, or false if the **end** of the collection has been reached. Prior to the first call to **MoveNext( )**, the value of **Current** is undefined. (Conceptually, prior to the first call to **MoveNext( )**, the enumerator refers to the nonexistent element that is just before the first element. Thus, you must call **MoveNext( )** to move to the first element.)

You can reset the enumerator to the start of the collection by calling **Reset( )**, shown here:

```
void Reset( )
```

After calling **Reset( )**, enumeration will again begin at the start of the collection. Thus, you must call **MoveNext( )** before obtaining the first element.

In **IEnumerator<T>**, the methods **MoveNext( )** and **Reset( )** work in the same way.

Two other points: First, you cannot use an enumerator to change the collection that it is enumerating. Thus, enumerators are read-only relative to the collection. Second, any change to the collection under enumeration invalidates the enumerator.

Using an Enumerator

Before you can access a collection through an enumerator, you must obtain one. Each of the collection classes provides a **GetEnumerator( )** method that returns an enumerator to the start of the collection. Using this enumerator, you can access each element in the collection, one element at a time. In general, to use an enumerator to cycle through the contents of a collection, follow these steps:

1. Obtain an enumerator to the start of the collection by calling the collection's **GetEnumerator( )** method.
2. Set up a loop that makes a call to **MoveNext( )**. Have the loop iterate as long as **MoveNext( )** returns true.
3. Within the loop, obtain each element through **Current**.

Here is an example that implements these steps. It uses an **ArrayList**, but the general principles apply to any type of collection, including the generic collections.

```
// Demonstrate an enumerator.

using System;
using System.Collections;

class EnumeratorDemo {
  static void Main() {
    ArrayList list = new ArrayList(1);

    for(int i=0; i < 10; i++)
      list.Add(i);

    // Use enumerator to access list.
    IEnumerator etr = list.GetEnumerator();
    while(etr.MoveNext())
      Console.Write(etr.Current + " ");

    Console.WriteLine();

    // Re-enumerate the list.
    etr.Reset();
    while(etr.MoveNext())
      Console.Write(etr.Current + " ");

    Console.WriteLine();
  }
}
```

The output is shown here:

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

In general, when you need to cycle through a collection, a **foreach** loop is more convenient to use than an enumerator. However, an enumerator gives you a little extra control by allowing you to reset the enumerator at will.

Using the IDictionaryEnumerator

When using a non-generic **IDictionary**, such as **Hashtable**, you will use an **IDictionaryEnumerator** instead of an **IEnumerator** when cycling through the collection. The **IDictionaryEnumerator** inherits **IEnumerator** and adds three properties. The first is

DictionaryEntry Entry { get; }

Entry obtains the next key/value pair from the enumerator in the form of a **DictionaryEntry** structure. Recall that **DictionaryEntry** defines two properties, called **Key** and **Value**, which can be used to access the key or value contained within the entry. The other two properties defined by **IDictionaryEnumerator** are shown here:

```
object Key { get; }
object Value { get; }
```

These allow you to access the key or value directly.

An **IDictionaryEnumerator** is used just like a regular enumerator, except that you will obtain the current value through the **Entry**, **Key**, or **Value** properties rather than **Current**. Thus, after obtaining an **IDictionaryEnumerator**, you must call **MoveNext( )** to obtain the first element. Continue to call **MoveNext( )** to obtain the rest of the elements in the collection. **MoveNext( )** returns false when there are no more elements.

Here is an example that enumerates the elements in a **Hashtable** through an **IDictionaryEnumerator**:

```
// Demonstrate IDictionaryEnumerator.

using System;
using System.Collections;

class IDicEnumDemo {
  static void Main() {
    // Create a hash table.
    Hashtable ht = new Hashtable();

    // Add elements to the table.
    ht.Add("Tom", "555-3456");
    ht.Add("Mary", "555-9876");
    ht.Add("Todd", "555-3452");
    ht.Add("Ken", "555-7756");

    // Demonstrate enumerator.
    IDictionaryEnumerator etr = ht.GetEnumerator();
    Console.WriteLine("Display info using Entry.");
    while(etr.MoveNext())
     Console.WriteLine(etr.Entry.Key + ": " +
                       etr.Entry.Value);

    Console.WriteLine();

    Console.WriteLine("Display info using Key and Value directly.");
    etr.Reset();
    while(etr.MoveNext())
     Console.WriteLine(etr.Key + ": " +
                       etr.Value);

  }
}
```

The output is shown here:

```
Display info using Entry.
Ken: 555-7756
Mary: 555-9876
Tom: 555-3456
Todd: 555-3452

Display info using Key and Value directly.
Ken: 555-7756
Mary: 555-9876
Tom: 555-3456
Todd: 555-3452
```


Implementing IEnumerable and IEnumerator

As mentioned earlier, normally it is easier (and better) to use a **foreach** loop to cycle through a collection than it is to explicitly use **IEnumerator** methods. However, understanding the operation of these interfaces is important for another reason: If you want to create a class that contains objects that can be enumerated via a **foreach** loop, then that class must implement **IEnumerator**. It must also implement **IEnumerable**. In other words, to enable an object of a class that you create to be used in a **foreach** loop, you must implement **IEnumerator** and **IEnumerable**, using either their generic or non-generic form. Fortunately, because these interfaces are so small, they are easy to implement.

NOTE *Technically, in order for a class to be used with a* ***foreach*** *loop, it does not actually have to specify* ***IEnumerator*** *or* ***IEnumerable*** *as implemented interfaces. It does, however, have to provide their methods, which are* ***GetEnumerator( )***, ***Reset( )***, ***MoveNext( )***, *and the property* ***Current****. However, not specifying these interfaces reduces the usability of the class in a mixed-language environment.*

Here is an example that implements the non-generic versions of **IEnumerable** and **IEnumerator** so that the contents of the array encapsulated within **MyClass** can be enumerated:

```
// Implement IEnumerable and IEnumerator.
using System;
using System.Collections;

class MyClass : IEnumerator, IEnumerable {
  char[] chrs = { 'A', 'B', 'C', 'D' };
  int idx = -1;

  // Implement IEnumerable.
  public IEnumerator GetEnumerator() {
    return this;
  }

  // The following methods implement IEnumerator.

  // Return the current object.
  public object Current {
    get {
```

```
      return chrs[idx];
    }
  }

  // Advance to the next object.
  public bool MoveNext() {
    if(idx == chrs.Length-1) {
      Reset(); // reset enumerator at the end
      return false;
    }

    idx++;
    return true;
  }

  // Reset the enumerator to the start.
  public void Reset() { idx = -1; }
}

class EnumeratorImplDemo {
  static void Main() {
    MyClass mc = new MyClass();

    // Display the contents of mc.
    foreach(char ch in mc)
      Console.Write(ch + " ");

    Console.WriteLine();

    // Display the contents of mc, again.
    foreach(char ch in mc)
      Console.Write(ch + " ");

    Console.WriteLine();
  }
}
```

Here is the output:

```
A B C D
A B C D
```

In the program, first examine **MyClass**. It encapsulates a small **char** array that contains the characters A through D. An index into this array is stored in **idx**, which is initialized to –1. **MyClass** then implements both **IEnumerator** and **IEnumerable**. **GetEnumerator( )** returns a reference to the enumerator, which in this case is the current object. The **Current** property returns the next character in the array, which is the object at **idx**. The **MoveNext( )** method advances **idx** to the next location. It returns false if the end of the collection has been reached and true otherwise. **Reset( )** sets **idx** to –1. Recall that an enumerator is undefined until after the first call to **MoveNext( )**. Thus, in a **foreach** loop, **MoveNext( )** is automatically called before **Current**. This is why *idx* must initially be –1; it is advanced to zero when the **foreach** loop begins. A generic implementation would work in a similar fashion.

Inside **Main( )**, an object of type **MyClass** called **mc** is created and the contents of the object are twice displayed by use of a **foreach** loop.

Using Iterators

As the preceding example shows, it is not difficult to implement **IEnumerator** and **IEnumerable**. However, it can be made even easier through the use of an *iterator.* An iterator is a method, operator, or accessor that returns the members of a set of objects, one member at a time, from start to finish. For example, assuming some array that has five elements, then an iterator for that array will return those five elements, one at a time. By implementing an iterator, you make it possible for an object of a class to be used in a **foreach** loop.

Let's begin with an example of a simple iterator. The following program is a modified version of the preceding program that uses an iterator rather than explicitly implementing **IEnumerator** and **IEnumerable**.


```
// A simple example of an iterator.

using System;
using System.Collections;

class MyClass {
  char[] chrs = { 'A', 'B', 'C', 'D' };

  // This iterator returns the characters
  // in the chrs array.
  public IEnumerator GetEnumerator() {
    foreach(char ch in chrs)
      yield return ch;
  }
}

class ItrDemo {
  static void Main() {
    MyClass mc = new MyClass();

    foreach(char ch in mc)
      Console.Write(ch + " ");

    Console.WriteLine();
  }
}
```

The output is shown here:

```
A B C D
```

As you can see, the contents of **mc.chrs** were enumerated.

Let's examine this program carefully. First, notice that **MyClass** does not specify **IEnumerator** as an implemented interface. When creating an iterator, the compiler automatically implements this interface for you. Second, pay special attention to the **GetEnumerator( )** method, which is shown again here for your convenience.

```
// This iterator returns the characters
// in the chrs array.
public IEnumerator GetEnumerator() {
  foreach(char ch in chrs)
    yield return ch;
}
```

This is the iterator for **MyClass**. Notice that it implicitly implements the **GetEnumerator( )** method defined by **IEnumerable**. Now, look at the body of the method. It contains a **foreach** loop that returns the elements in **chrs**. It does this through the use of a **yield return** statement. The **yield return** statement returns the next object in the collection, which in this case is the next character in **chrs**. This feature enables **mc** (a **MyClass** object) to be used within the **foreach** loop inside **Main( )**.

The term **yield** is a *contextual keyword* in the C# language. This means that it only has special meaning inside an iterator block. Outside of an iterator, **yield** can be used like any other identifier.

One important point to understand is that an iterator does not need to backed by an array or other type of collection. It simply must return the next element in a group of elements. This means the elements can be dynamically constructed using an algorithm. For example, here is a version of the previous program that returns all uppercase letters in the alphabet. Instead of using an array, it generates the letters using a **for** loop.

```
// Iterated values can be dynamically constructed.

using System;
using System.Collections;

class MyClass {
  char ch = 'A';

  // This iterator returns the letters of the alphabet.
  public IEnumerator GetEnumerator() {
    for(int i=0; i < 26; i++)
      yield return (char) (ch + i);
  }
}

class ItrDemo2 {
  static void Main() {
    MyClass mc = new MyClass();

    foreach(char ch in mc)
      Console.Write(ch + " ");

    Console.WriteLine();
  }
}
```

The output is shown here:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Stopping an Iterator

You can stop an iterator early by using this form of the **yield** statement:

```
yield break;
```

When this statement executes, the iterator signals that the end of the collection has been reached, which effectively stops the iterator.

The following program modifies the preceding program so that it displays only the first ten letters in the alphabet.

```
// Use yield break.

using System;
using System.Collections;

class MyClass {
  char ch = 'A';

  // This iterator returns the first 10
  // letters of the alphabet.
  public IEnumerator GetEnumerator() {
    for(int i=0; i < 26; i++) {
      if(i == 10) yield break; // stop iterator early
      yield return (char) (ch + i);
    }
  }
}

class ItrDemo3 {
  static void Main() {
    MyClass mc = new MyClass();

    foreach(char ch in mc)
      Console.Write(ch + " ");

    Console.WriteLine();
  }
}
```

The output is shown here:

```
A B C D E F G H I J
```

Using Multiple yield Directives

You can have more than one **yield** statement in an iterator. However, each **yield** must return the next element in the collection. For example, consider this program:

```
// Multiple yield statements are allowed.

using System;
using System.Collections;
```

```
class MyClass {
  // This iterator returns the letters
  // A, B, C, D, and E.
  public IEnumerator GetEnumerator() {
    yield return 'A';
    yield return 'B';
    yield return 'C';
    yield return 'D';
    yield return 'E';
  }
}

class ItrDemo5 {
  static void Main() {
    MyClass mc = new MyClass();

    foreach(char ch in mc)
      Console.Write(ch + " ");

    Console.WriteLine();
  }
}
```

The output is shown here:

```
A B C D E
```

Inside **GetEnumerator( )**, five yield statements occur. The important thing to understand is that they are executed one at a time, in order, each time another element in the collection is obtained. Thus, each time through the **foreach** loop in **Main( )**, one character is returned.

Creating a Named Iterator

Although the preceding examples have shown the easiest way to implement an iterator, there is an alternative: the named iterator. In this approach, you create a method, operator, or accessor that returns a reference to an **IEnumerable** object. Your code will use this object to supply the iterator. A named iterator is a method with the following general form:

```
public IEnumerable itr-name(param-list) {
  // ...
  yield return obj;

}
```

Here, *itr-name* is the name of the method, *param-list* specifies zero or more parameters that will be passed to the iterator method, and *obj* is the next object returned by the iterator. Once you have created a named iterator, you can use it anywhere that an iterator is needed. For example, you can use the named iterator to control a **foreach** loop.

Named iterators are very useful in some circumstances because they allow you to pass arguments to the iterator that control what elements are obtained. For example, you might pass the iterator the beginning and ending points of a range of elements to iterate. This form of iterator can also be overloaded, further adding to its flexibility. The following program

illustrates two ways that a named iterator can be used to obtain elements. One enumerates a range of elements given the endpoints. The other enumerates the elements beginning at the start of the sequence and ending at the specified stopping point.

```
// Use named iterators.

using System;
using System.Collections;

class MyClass {
  char ch = 'A';

  // This iterator returns the letters
  // of the alphabet, beginning at A and
  // stopping at the specified stopping point.
  public IEnumerable MyItr(int end) {
    for(int i=0; i < end; i++)
      yield return (char) (ch + i);
  }

  // This iterator returns the specified
  // range of letters.
  public IEnumerable MyItr(int begin, int end) {
    for(int i=begin; i < end; i++)
      yield return (char) (ch + i);
  }
}

class ItrDemo4 {
  static void Main() {
    MyClass mc = new MyClass();

    Console.WriteLine("Iterate the first 7 letters:");
    foreach(char ch in mc.MyItr(7))
      Console.Write(ch + " ");

    Console.WriteLine("\n");


    Console.WriteLine("Iterate letters from F to L:");
    foreach(char ch in mc.MyItr(5, 12))
      Console.Write(ch + " ");

    Console.WriteLine();
  }
}
```

The output is shown here:

```
Iterate the first 7 letters:
A B C D E F G

Iterate letters from F to L:
F G H I J K L
```

Creating a Generic Iterator

The preceding examples of iterators have been non-generic, but it is, of course, also possible to create generic iterators. Doing so is quite easy: Simply return an object of the generic **IEnumerator<T>** or **IEnumerable<T>** type. Here is an example that creates a generic iterator:

```
// A simple example of a generic iterator.

using System;
using System.Collections.Generic;

class MyClass<T> {
  T[] array;

  public MyClass(T[] a) {
    array = a;
  }

  // This iterator returns the characters
  // in the chrs array.
  public IEnumerator<T> GetEnumerator() {
    foreach(T obj in array)
      yield return obj;
  }
}

class GenericItrDemo {
  static void Main() {
    int[] nums = { 4, 3, 6, 4, 7, 9 };
    MyClass<int> mc = new MyClass<int>(nums);

    foreach(int x in mc)
      Console.Write(x + " ");

    Console.WriteLine();


    bool[] bVals = { true, true, false, true };
    MyClass<bool> mc2 = new MyClass<bool>(bVals);

    foreach(bool b in mc2)
      Console.Write(b + " ");

    Console.WriteLine();
  }
}
```

The output is shown here:

```
4 3 6 4 7 9
True True False True
```

In this example, the array containing the objects to be iterated is passed to **MyClass** through its constructor. The type of the array is specified as a type argument to **MyClass**.

The **GetEnumerator( )** method operates on data of type **T** and returns an **IEnumerator<T>** enumerator. Thus, the iterator defined by **MyClass** can enumerate any type of data.

Collection Initializers

C# 3.0 includes a new feature called the *collection initializer,* which makes it easier to initialize certain collections. Instead of having to explicitly call **Add( )**, you can specify a list of initializers when a collection is created. When this is done, the compiler automatically calls **Add( )** for you, using these values. The syntax is similar to an array initialization. Here is an example. It creates a **List<char>** that is initialized by the characters C, A, E, B, D, and F.

```
List<char> lst = new List<char>() { 'C', 'A', 'E', 'B', 'D', 'F' };
```

After this statement executes, **lst.Count** will equal 6, because there are six initializers, and this **foreach** loop

```
foreach(ch in lst)
  Console.Write(ch + " ");
```

will display

```
C A E B D F
```

When using a collection such as **LinkedList<TK, TV>** that store key/value pairs, you will need to supply pairs of initializers, as shown here:

```
SortedList<int, string> lst =
  new SortedList<int, string>() { {1, "One"}, {2, "Two" }, {3, "Three"} };
```

The compiler passes each group of values as arguments to **Add( )**. Thus, the first pair of initializers is translated into a call to **Add(1, "One")** by the compiler.

Because the compiler automatically calls **Add( )** to add initializers to a collection, collection initializers can be used only with collections that support a public implementation of **Add( )**. Therefore, collection initializers cannot be used with the **Stack**, **Stack<T>**, **Queue**, or **Queue<T>** collections because they don't support **Add( )**. You also can't use a collection initializer with a collection such as **LinkedList<T>**, which provides **Add( )** as an explicit interface implementation.

This page intentionally left blank

CHAPTER 25

Networking Through the Internet Using System.Net

C# is a language designed for the modern computing environment, of which the Internet is, obviously, an important part. A main design criteria for C# was, therefore, to include those features necessary for accessing the Internet. Although earlier languages, such as C and C++, could be used to access the Internet, support server-side operations, download files, and obtain resources, the process was not as streamlined as most programmers would like. C# remedies that situation. Using standard features of C# and the .NET Framework, it is easy to "Internet-enable" your applications and write other types of Internet-based code.

Networking support is contained in several namespaces defined by the .NET Framework. The primary namespace for networking is **System.Net**. It defines a large number of high-level, easy-to-use classes that support the various types of operations common to the Internet. Several namespaces nested under **System.Net** are also provided. For example, low-level networking control through sockets is found in **System.Net.Sockets**. Mail support is found in **System.Net.Mail**. Support for secure network streams is found in **System.Net.Security**. Several other nested namespaces provide additional functionality. Another important networking-related namespace is **System.Web**. It (and its nested namespaces) support ASP.NET-based network applications.

Although the .NET Framework offers great flexibility and many options for networking, for many applications the functionality provided by **System.Net** is a best choice. It offers both convenience and ease-of-use. For this reason, **System.Net** is the namespace we will be using in this chapter.

The System.Net Members

System.Net is a large namespace that contains many members. It is far beyond the scope of this chapter to discuss them all or to discuss all aspects related to Internet programming. (In fact, an entire book is needed to fully cover networking and C#'s support for it in detail.) However, it is worthwhile to list the members of **System.Net** so you have an idea of what is available for your use.

The classes defined by **System.Net** are shown here:

AuthenticationManager	Authorization
Cookie	CookieCollection
CookieContainer	CookieException
CredentialCache	Dns
DnsPermission	DnsPermissionAttribute
DownloadDataCompletedEventArgs	DownloadProgressChangedEventArgs
DownloadStringCompletedEventArgs	EndPoint
EndpointPermission	FileWebRequest
FileWebResponse	FtpWebRequest
FtpWebResponse	HttpListener
HttpListenerBasicIdentity	HttpListenerContext
HttpListenerException	HttpListenerPrefixCollection
HttpListenerRequest	HttpListenerResponse
HttpVersion	HttpWebRequest
HttpWebResponse	IPAddress
IPEndPoint	IPEndPointCollection
IPHostEntry	IrDAEndPoint
NetworkCredential	OpenReadCompletedEventArgs
OpenWriteCompletedEventArgs	ProtocolViolationException
ServicePoint	ServicePointManager
SocketAddress	SocketPermission
SocketPermissionAttribute	UploadDataCompletedEventArgs
UploadFileCompletedEventArgs	UploadProgressChangedEventArgs
UploadStringCompletedEventArgs	UploadValuesCompletedEventArgs
WebClient	WebException
WebHeaderCollection	WebPermission
WebPermissionAttribute	WebProxy
WebRequest	WebRequestMethods
WebRequestMethods.File	WebRequestMethods.Ftp
WebRequestMethods.Http	WebResponse

System.Net defines the following interfaces:

IAuthenticationModule	ICertificatePolicy	ICredentialPolicy
ICredentials	ICredentialsByHost	IWebProxy
IWebProxyScript	IWebRequestCreate	

It defines these enumerations:

AuthenticationSchemes	DecompressionMethods	FtpStatusCode
HttpRequestHeader	HttpResponseHeader	HttpStatusCode
NetworkAccess	SecurityProtocolType	TransportType
WebExceptionStatus		

System.Net also defines several delegates.

Although **System.Net** defines many members, only a few are needed to accomplish most common Internet programming tasks. At the core of networking are the abstract classes **WebRequest** and **WebResponse**. These classes are inherited by classes that support a specific network protocol. (A *protocol* defines the rules used to send information over a network.) For example, the derived classes that support the standard HTTP protocol are **HttpWebRequest** and **HttpWebResponse**.

Even though **WebRequest** and **WebResponse** are easy to use, for some tasks you can employ an even simpler approach based on **WebClient**. For example, if you only need to upload or download a file, then **WebClient** is often the best way to accomplish it.

Uniform Resource Identifiers

Fundamental to Internet programming is the Uniform Resource Identifier (URI). A *URI* describes the location of some resource on the network. A URI is also commonly called a *URL*, which is short for *Uniform Resource Locator*. Because Microsoft uses the term *URI* when describing the members of **System.Net**, this book will do so, too. You are no doubt familiar with URIs because you use one every time you enter an address into your Internet browser.

A URI has the following general form:

Protocol://ServerID/FilePath?Query

Protocol specifies the protocol being used, such as HTTP. *ServerID* identifies the specific server, such as mhprofessional.com or HerbSchildt.com. *FilePath* specifies the path to a specific file. If *FilePath* is not specified, the default page at the specified *ServerID* is obtained. Finally, *Query* specifies information that will be sent to the server. *Query* is optional. In C#, URIs are encapsulated by the **Uri** class, which is examined later in this chapter.

Internet Access Fundamentals

The classes contained in **System.Net** support a request/response model of Internet interaction. In this approach, your program, which is the client, requests information from the server and then waits for the response. For example, as a request, your program might send to the server the URI of some website. The response that you will receive is the hypertext associated with that URI. This request/response approach is both convenient and simple to use because most of the details are handled for you.

The hierarchy of classes topped by **WebRequest** and **WebResponse** implement what Microsoft calls *pluggable protocols*. As most readers know, there are several different types of network communication protocols. The most common for Internet use is HyperText Transfer Protocol (HTTP). Another is File Transfer Protocol (FTP). When a URI is constructed, the

prefix of the URI specifies the protocol. For example, http://www.HerbSchildt.com uses the prefix *http*, which specifies hypertext transfer protocol.

As mentioned earlier, **WebRequest** and **WebResponse** are abstract classes that define the general request/response operations that are common to all protocols. From them are derived concrete classes that implement specific protocols. Derived classes register themselves, using the static method **RegisterPrefix( )**, which is defined by **WebRequest**. When you create a **WebRequest** object, the protocol specified by the URI's prefix will automatically be used, if it is available. The advantage of this "pluggable" approach is that most of your code remains the same no matter what type of protocol you are using.

The .NET runtime automatically defines the HTTP, HTTPS, file, and FTP protocols. Thus, if you specify a URI that uses the HTTP prefix, you will automatically receive the HTTP-compatible class that supports it. If you specify a URI that uses the FTP prefix, you will automatically receive the FTP-compatible class that supports it.

Because HTTP is the most commonly used protocol, it is the only one discussed in this chapter. (The same techniques, however, will apply to all supported protocols.) The classes that support HTTP are **HttpWebRequest** and **HttpWebResponse**. These classes inherit **WebRequest** and **WebResponse** and add several members of their own, which apply to the HTTP protocol.

System.Net supports both synchronous and asynchronous communication. For many Internet uses, synchronous transactions are the best choice because they are easy to use. With synchronous communications, your program sends a request and then waits until the response is received. For some types of high-performance applications, asynchronous communication is better. Using the asynchronous approach, your program can continue processing while waiting for information to be transferred. However, asynchronous communications are more difficult to implement. Furthermore, not all programs benefit from an asynchronous approach. For example, often when information is needed from the Internet, there is nothing to do until the information is received. In cases like this, the potential gains from the asynchronous approach are not realized. Because synchronous Internet access is both easier to use and more universally applicable, it is the only type examined in this chapter.

Since **WebRequest** and **WebResponse** are at the heart of **System.Net**, they will be examined next.

WebRequest

The **WebRequest** class manages a network request. It is abstract because it does not implement a specific protocol. It does, however, define those methods and properties common to all requests. The methods defined by **WebRequest** that support synchronous communications are shown in Table 25-1. The properties defined by **WebRequest** are shown in Table 25-2. The default values for the properties are determined by derived classes. **WebRequest** defines no public constructors.

To send a request to a URI, you must first create an object of a class derived from **WebRequest** that implements the desired protocol. This is done by calling **Create( )**, which is a **static** method defined by **WebRequest**. **Create( )** returns an object of a class that inherits **WebRequest** and implements a specific protocol.

Method	Description
public static WebRequest Create(string *uri*)	Creates a **WebRequest** object for the URI specified by the string passed by *uri*. The object returned will implement the protocol specified by the prefix of the URI. Thus, the object will be an instance of a class that inherits **WebRequest**. A **NotSupportedException** is thrown if the requested protocol is not available. A **UriFormatException** is thrown if the URI format is invalid.
public static WebRequest Create(Uri *uri*)	Creates a **WebRequest** object for the URI specified by *uri*. The object returned will implement the protocol specified by the prefix of the URI. Thus, the object will be an instance of a class that inherits **WebRequest**. A **NotSupportedException** is thrown if the requested protocol is not available.
public virtual Stream GetRequestStream( )	Returns an output stream associated with the previously requested URI.
public virtual WebResponse GetResponse( )	Sends the previously created request and waits for a response. When a response is received, it is returned as a **WebReponse** object. Your program will use this object to obtain information from the specified URI.

Table 25-1 The Methods Defined by **WebRequest** that Support Synchronous Communications

Property	Description
public AuthenticationLevel AuthenticationLevel(get; set; }	Obtains or sets the authentication level.
public virtual RequestCachePolicy CachePolicy { get; set; }	Obtains or sets the cache policy, which controls when a response can be obtained from the cache.
public virtual string ConnectionGroupName { get; set; }	Obtains or sets the connection group name. Connection groups are a way of creating a set of requests. They are not needed for simple Internet transactions.
public virtual long ContentLength { get; set; }	Obtains or sets the length of the content.
public virtual string ContentType { get; set; }	Obtains or sets the description of the content.
public virtual ICredentials Credentials { get; set; }	Obtains or sets credentials.
public static RequestCachePolicy DefaultCachePolicy { get; set; }	Obtains or sets the default cache policy, which controls when a request can be obtained from the cache.

Table 25-2 The Properties Defined by **WebRequest**

Property	Description
public static IWebProxy DefaultWebProxy { get; set; }	Obtains or sets the default proxy.
public virtual WebHeaderCollection Headers{ get; set; }	Obtains or sets a collection of the headers.
public TokenImpersonationLevel ImpersonationLevel { get; set; }	Obtains or sets the impersonation level.
public virtual string Method { get; set; }	Obtains or sets the protocol.
public virtual bool PreAuthenticate { get; set; }	If true, authentication information is included when the request is sent. If false, authentication information is provided only when requested by the URI.
public virtual IWebProxy Proxy { get; set; }	Obtains or sets the proxy server. This applies only to environments in which a proxy server is used.
public virtual Uri RequestUri { get; }	Obtains the URI of the request.
public virtual int Timeout { get; set; }	Obtains or sets the number of milliseconds that a request will wait for a response. To wait forever, use **Timeout.Infinite**.
public virtual bool UseDefaultCredential { get; set; }	Obtains or sets a value that determines if default credentials are used for authentication. If true, the default credentials (i.e., those of the user) are used. They are not used if false.

TABLE 25-2 The Properties Defined by **WebRequest** *(continued)*

WebResponse

WebResponse encapsulates a response that is obtained as the result of a request. **WebResponse** is an abstract class. Inheriting classes create specific, concrete versions of it that support a protocol. A **WebResponse** object is normally obtained by calling the **GetResponse( )** method defined by **WebRequest**. This object will be an instance of a concrete class derived from **WebResponse** that implements a specific protocol. The methods defined by **WebResponse** that are most commonly used are shown in Table 25-3. The properties defined by **WebResponse** are shown in Table 25-4. The values of these properties are set based on each individual response. **WebResponse** defines no public constructors.

HttpWebRequest and HttpWebResponse

The classes **HttpWebRequest** and **HttpWebResponse** inherit the **WebRequest** and **WebResponse** classes and implement the HTTP protocol. In the process, both add several properties that give you detailed information about an HTTP transaction. Some of these

Method	Description
public virtual void Close( )	Closes the response. It also closes the response stream returned by **GetResponseStream( )**.
public virtual Stream GetResponseStream( )	Returns an input stream connected to the requested URI. Using this stream, data can be read from the URI.

TABLE 25-3 Commonly Used Methods Defined by **WebResponse**

properties are used later in this chapter. However, for simple Internet operations, you will not often need to use these extra capabilities.

A Simple First Example

Internet access centers around **WebRequest** and **WebResponse**. Before we examine the process in detail, it will be useful to see an example that illustrates the request/response approach to Internet access. After you see these classes in action, it is easier to understand why they are organized as they are.

The following program performs a simple, yet very common, Internet operation. It obtains the hypertext contained at a specific website. In this case, the content of McGraw-Hill.com is obtained, but you can substitute any other website. The program displays the

Property	Description
public virtual long ContentLength { get; set; }	Obtains or sets the length of the content being received. This will be –1 if the content length is not available.
public virtual string ContentType { get; set; }	Obtains or sets a description of the content.
public virtual WebHeaderCollection Headers { get; }	Obtains a collection of the headers associated with the URI.
public virtual bool IsFromCache { get; }	If the response came from the cache, this property is true. It is false if the response was delivered over the network.
public virtual bool IsMutuallyAuthenticated { get; }	If the client and server are both authenticated, then this property is true. It is false otherwise.
public virtual Uri ResponseUri { get; }	Obtains the URI that generated the response. This may differ from the one requested if the response was redirected to another URI.

TABLE 25-4 The Properties Defined by **WebResponse**

hypertext on the screen in chunks of 400 characters, so you can see what is being received before it scrolls off the screen.

```
// Access a website.

using System;
using System.Net;
using System.IO;

class NetDemo {
  static void Main() {
    int ch;

    // First, create a WebRequest to a URI.
    HttpWebRequest req = (HttpWebRequest)
           WebRequest.Create("http://www.McGraw-Hill.com");

    // Next, send that request and return the response.
    HttpWebResponse resp = (HttpWebResponse)
           req.GetResponse();

    // From the response, obtain an input stream.
    Stream istrm = resp.GetResponseStream();


    /* Now, read and display the html present at
       the specified URI. So you can see what is
       being displayed, the data is shown
       400 characters at a time. After each 400
       characters are displayed, you must press
       ENTER to get the next 400. */

    for(int i=1; ; i++) {
      ch =  istrm.ReadByte();
      if(ch == -1) break;
      Console.Write((char) ch);
      if((i%400)==0) {
        Console.Write("\nPress Enter.");
        Console.ReadLine();
      }
    }

    // Close the response. This also closes istrm.
    resp.Close();
  }
}
```

The first part of the output is shown here. (Of course, over time this content will differ from that shown here.)

```
<html>
<head>
<title>Home - The McGraw-Hill Companies</title>
<meta name="keywords" content="McGraw-Hill Companies,McGraw-Hill, McGraw Hill,
Aviation Week, BusinessWeek, Standard and Poor's, Standard & Poor's,CTB/Mc-
```

```
Graw-Hill,Glencoe/McGraw-Hill,The Grow Network/McGraw-Hill,Macmillan/McGraw-
Hill,McGraw-Hill Contemporary,McGraw-Hill Digital Learning,McGraw-Hill Professional
Development,SRA/McGraw-Hi
Press Enter.
ll,Wright Group/McGraw-Hill,McGraw-Hill Higher Education,McGraw-Hill/Irwin,McGraw-
Hill/Primis Custom Publishing,McGraw-Hill/Ryerson,Tata/McGraw-Hill,McGraw-Hill
Interamericana,Open University Press, Healthcare Information Group, Platts, McGraw-
Hill Construction, Information & Media Services" />
<meta name="description" content="The McGraw-Hill Companies Corporate Website." />
<meta http-equiv="Con
Press Enter.
tent-Type" content="text/html; charset=iso-8859-1">
<META HTTP-EQUIV="Refresh" CONTENT="900">
<META HTTP-EQUIV="EXPIRES" CONTENT="-1">
<META HTTP-EQUIV="Pragma" CONTENT="no-cache">
<link rel="stylesheet" href="stylesheet.css" type="text/css" media="screen,projecti
on">
<link rel="stylesheet" href="print.css" type"text/css" media="print">

<script language="JavaScript1.2" src="scripts.js"></script>

Press Enter.
```

This is part of the hypertext associated with the McGraw-Hill.com website. Because the program simply displays the content character-by-character, it is not formatted as it would be by a browser; it is displayed in its raw form.

Let's examine this program line-by-line. First, notice that the **System.Net** namespace is used. As explained, this is the namespace that contains the networking classes. Also notice that **System.IO** is included. This namespace is needed because the information from the website is read using a **Stream** object.

The program begins by creating a **WebRequest** object that contains the desired URI. Notice that the **Create( )** method, rather than a constructor, is used for this purpose. **Create( )** is a **static** member of **WebRequest**. Even though **WebRequest** is an abstract class, it is still possible to call a **static** method of that class. **Create( )** returns a **WebRequest** object that has the proper protocol "plugged in," based on the protocol prefix of the URI. In this case, the protocol is HTTP. Thus, **Create( )** returns an **HttpWebRequest** object. Of course, its return value must still be cast to **HttpWebRequest** when it is assigned to the **HttpWebRequest** reference called **req**. At this point, the request has been created, but not yet sent to the specified URI.

To send the request, the program calls **GetResponse( )** on the **WebRequest** object. After the request has been sent, **GetResponse( )** waits for a response. Once a response has been received, **GetResponse( )** returns a **WebResponse** object that encapsulates the response. This object is assigned to **resp**. Since, in this case, the response uses the HTTP protocol, the result is cast to **HttpWebResponse**. Among other things, the response contains a stream that can be used to read data from the URI.

Next, an input stream is obtained by calling **GetResponseStream( )** on **resp**. This is a standard **Stream** object, having all of the attributes and features of any other input stream. A reference to the stream is assigned to **istrm**. Using **istrm**, the data at the specified URI can be read in the same way that a file is read.

Next, the program reads the data from McGraw-Hill.com and displays it on the screen. Because there is a lot of information, the display pauses every 400 characters and waits for you to press ENTER. This way the first part of the information won't simply scroll off the

screen. Notice that the characters are read using **ReadByte( )**. Recall that this method returns the next byte from the input stream as an **int**, which must be cast to **char**. It returns –1 when the end of the stream has been reached.

Finally, the response is closed by calling **Close( )** on **resp**. Closing the response stream automatically closes the input stream, too. It is important to close the response between each request. If you don't, it is possible to exhaust the network resources and prevent the next connection.

Before leaving this example, one other important point needs to be made: It was not actually necessary to use an **HttpWebRequest** or **HttpWebResponse** object to display the hypertext received from the server. Because the preceding program did not use any HTTP-specific features, the standard methods defined by **WebRequest** and **WebResponse** were sufficient to handle this task. Thus, the calls to **Create( )** and **GetResponse( )** could have been written like this:

```
// First, create a WebRequest to a URI.
WebRequest req =  WebRequest.Create("http://www.McGraw-Hill.com");

// Next, send that request and return the response.
WebResponse resp =  req.GetResponse();
```

In cases in which you don't need to employ a cast to a specific type of protocol implementation, it is better to use **WebRequest** and **WebResponse** because it allows protocols to be changed with no impact on your code. However, since all of the examples in this chapter will be using HTTP, and a few will be using HTTP-specific features, the programs will use **HttpWebRequest** and **HttpWebResponse**.

Handling Network Errors

Although the program in the preceding section is correct, it is not resilient. Even the simplest network error will cause it to end abruptly. Although this isn't a problem for the example programs shown in this chapter, it is something that must be avoided in real-world applications. To fully handle all network exceptions that the program might generate, you must monitor calls to **Create( )**, **GetResponse( )**, and **GetResponseStream( )**. It is important to understand that the exceptions that can be generated depend upon the protocol being used. The following discussion describes the errors possible when using HTTP.

Exceptions Generated by Create( )

The **Create( )** method defined by **WebRequest** can generate four exceptions. If the protocol specified by the URI prefix is not supported, then **NotSupportedException** is thrown. If the URI format is invalid, **UriFormatException** is thrown. If the user does not have the proper authorization, a **System.Security.SecurityException** will be thrown. **Create( )** can also throw an **ArgumentNullException** if it is called with a null reference, but this is not an error generated by networking.

Exceptions Generated by GetReponse( )

A number of errors can occur when obtaining an HTTP response by calling **GetResponse( )**. These are represented by the following exceptions: **InvalidOperationException**, **ProtocolViolationException**, **NotSupportedException**, and **WebException**. Of these, the one of most interest is **WebException**.

WebException has two properties that relate to network errors: **Response** and **Status**. You can obtain a reference to the **WebResponse** object inside an exception handler through the **Response** property. For the HTTP protocol, this object describes the error. It is defined like this:

```
public WebResponse Response { get; }
```

When an error occurs, you can use the **Status** property of **WebException** to find out what went wrong. It is defined like this:

```
public WebExceptionStatus Status {get; }
```

WebExceptionStatus is an enumeration that contains the following values:

CacheEntryNotFound	ConnectFailure	ConnectionClosed
KeepAliveFailure	MessageLengthLimitExceeded	NameResolutionFailure
Pending	PipelineFailure	ProtocolError
ProxyNameResolutionFailure	ReceiveFailure	RequestCanceled
RequestProhibitedByCachePolicy	RequestProhibitedByProxy	SecureChannelFailure
SendFailure	ServerProtocolViolation	Success
Timeout	TrustFailure	UnknownError

Once the cause of the error has been determined, your program can take appropriate action.

Exceptions Generated by GetResponseStream( )

For the HTTP protocol, the **GetResponseStream( )** method of **WebResponse** can throw a **ProtocolViolationException**, which, in general, means that some error occurred relative to the specified protocol. As it relates to **GetResponseStream( )**, it means that no valid response stream is available. An **ObjectDisposedException** will be thrown if the response has already been disposed. Of course, an **IOException** could occur while reading the stream, depending on how input is accomplished.

Using Exception Handling

The following program adds handlers for all possible network exceptions to the example shown earlier:

```
// Handle network exceptions.

using System;
using System.Net;
using System.IO;

class NetExcDemo {
  static void Main() {
    int ch;

    try {

      // First, create a WebRequest to a URI.
      HttpWebRequest req = (HttpWebRequest)
```

```
            WebRequest.Create("http://www.McGraw-Hill.com");

      // Next, send that request and return the response.
      HttpWebResponse resp = (HttpWebResponse)
            req.GetResponse();

      // From the response, obtain an input stream.
      Stream istrm = resp.GetResponseStream();

      /* Now, read and display the html present at
         the specified URI. So you can see what is
         being displayed, the data is shown
         400 characters at a time. After each 400
         characters are displayed, you must press
         ENTER to get the next 400. */

      for(int i=1; ; i++) {
        ch =  istrm.ReadByte();
        if(ch == -1) break;
        Console.Write((char) ch);
        if((i%400)==0) {
          Console.Write("\nPress Enter.");
          Console.ReadLine();
        }
      }

      // Close the response. This also closes istrm.
      resp.Close();

    } catch(WebException exc) {
      Console.WriteLine("Network Error: " + exc.Message +
                        "\nStatus code: " + exc.Status);
    } catch(ProtocolViolationException exc) {
      Console.WriteLine("Protocol Error: " + exc.Message);
    } catch(UriFormatException exc) {
      Console.WriteLine("URI Format Error: " + exc.Message);
    } catch(NotSupportedException exc) {
      Console.WriteLine("Unknown Protocol: " + exc.Message);
    } catch(IOException exc) {
      Console.WriteLine("I/O Error: " + exc.Message);
    } catch(System.Security.SecurityException exc) {
      Console.WriteLine("Security Exception: " + exc.Message);
    } catch(InvalidOperationException exc) {
      Console.WriteLine("Invalid Operation: " + exc.Message);
    }
  }
}
```

Now the exceptions that the networking methods might generate have been caught. For example, if you change the call to **Create( )** as shown here,

```
WebRequest.Create("http://www.McGraw-Hill.com/moonrocket");
```

and then recompile and run the program, you will see this output:

```
Network Error: The remote server returned an error: (404) Not Found.
Status code: ProtocolError
```

Since the McGraw-Hill.com website does not have a directory called "moonrocket," this URI is not found, as the output confirms.

To keep the examples short and uncluttered, most of the programs in this chapter will not contain full exception handling. However, your real-world applications must.

The Uri Class

In Table 25-1, notice that **WebRequest.Create( )** has two different versions. One accepts the URI as a string. This is the version used by the preceding programs. The other takes the URI as an instance of the **Uri** class, which is defined in the **System** namespace. The **Uri** class encapsulates a URI. Using **Uri**, you can construct a URI that can be passed to **Create( )**. You can also dissect a **Uri**, obtaining its parts. Although you don't need to use **Uri** for many simple Internet operations, you may find it valuable in more sophisticated situations.

Uri defines several constructors. Two commonly used ones are shown here:

public Uri(string *uri*)
public Uri(Uri *base*, string *rel*)

The first form constructs a **Uri** given a URI in string form. The second constructs a **Uri** by adding a relative URI specified by *rel* to an absolute base URI specified by *base*. An absolute URI defines a complete URI. A relative URI defines only the path.

Uri defines many fields, properties, and methods that help you manage URIs or that give you access to the various parts of a URI. Of particular interest are the properties shown here:

Property	Description
public string Host { get; }	Obtains the name of the server.
public string LocalPath { get; }	Obtains the local file path.
public string PathAndQuery { get; }	Obtains the absolute path and query string.
public int Port { get; }	Obtains the port number for the specified protocol. For HTTP, the port is 80.
public string Query { get; }	Obtains the query string.
public string Scheme { get; }	Obtains the protocol.

These properties are useful for breaking a URI into its constituent parts. The following program demonstrates their use:

```
// Use Uri.

using System;
using System.Net;
```

```
class UriDemo {
  static void Main() {

    Uri sample = new Uri("http://HerbSchildt.com/somefile.txt?SomeQuery");

    Console.WriteLine("Host: " + sample.Host);
    Console.WriteLine("Port: " + sample.Port);
    Console.WriteLine("Scheme: " + sample.Scheme);
    Console.WriteLine("Local Path: " + sample.LocalPath);
    Console.WriteLine("Query: " + sample.Query);
    Console.WriteLine("Path and query: " + sample.PathAndQuery);

  }
}
```

The output is shown here:

```
Host: HerbSchildt.com
Port: 80
Scheme: http
Local Path: /somefile.txt
Query: ?SomeQuery
Path and query: /somefile.txt?SomeQuery
```

Accessing Additional HTTP Response Information

When using **HttpWebResponse**, you have access to information other than the content of the specified resource. This information includes such things as the time the resource was last modified and the name of the server, and is available through various properties associated with the response. These properties, which include the six defined by **WebResponse**, are shown in Table 25-5. The following sections illustrate how to use representative samples.

Accessing the Header

You can access the header information associated with an HTTP response through the **Headers** property defined by **HttpWebResponse**. It is shown here:

```
public WebHeaderCollection Headers{ get; }
```

An HTTP header consists of pairs of names and values represented as strings. Each name/value pair is stored in a **WebHeaderCollection**. This specialized collection stores key/value pairs and can be used like any other collection. (See Chapter 24.) A **string** array of the names can be obtained from the **AllKeys** property. You can obtain the values associated with a name by calling the **GetValues( )** method. It returns an array of strings that contains the values associated with the header passed as an argument. **GetValues( )** is overloaded to accept a numeric index or the name of the header.

Property	Description
public string CharacterSet { get; }	Obtains the name of the character set being used.
public string ContentEncoding { get; }	Obtains the name of the encoding scheme.
public long ContentLength { get; }	Obtains the length of the content being received. This will be –1 if the content length is not available.
public string ContentType { get; }	Obtains a description of the content.
public CookieCollection Cookies { get; set; }	Obtains or sets a list of the cookies attached to the response.
public WebHeaderCollection Headers{ get; }	Obtains a collection of the headers attached to the response.
public bool IsFromCache { get; }	If the response came from the cache, this property is true. It is false if the response was delivered over the network.
public bool IsMutuallyAuthenticated { get; }	If the client and server are both authenticated, then this property is true. It is false otherwise.
public DateTime LastModified { get; }	Obtains the time at which the resource was last changed.
public string Method { get; }	Obtains a string that specifies the response method.
public Version ProtocolVersion { get; }	Obtains a **Version** object that describes the version of HTTP used in the transaction.
public Uri ReponseUri { get; }	Obtains the URI that generated the response. This may differ from the one requested if the response was redirected to another URI.
public string Server { get; }	Obtains a string that represents the name of the server.
public HttpStatusCode StatusCode { get; }	Obtains an **HttpStatusCode** object that describes the status of the transaction.
public string StatusDescription { get; }	Obtains a string that represents the status of the transaction in a human-readable form.

TABLE 25-5 The Properties Defined by **HttpWebResponse**

The following program displays headers associated with McGraw-Hill.com:

```
// Examine the headers.

using System;
using System.Net;
```

```
class HeaderDemo {
  static void Main() {

    // Create a WebRequest to a URI.
    HttpWebRequest req = (HttpWebRequest)
           WebRequest.Create("http://www.McGraw-Hill.com");

    // Send that request and return the response.
    HttpWebResponse resp = (HttpWebResponse)
           req.GetResponse();

    // Obtain a list of the names.
    string[] names = resp.Headers.AllKeys;

    // Display the header name/value pairs.
    Console.WriteLine("{0,-20}{1}\n", "Name", "Value");
    foreach(string n in names) {
      Console.Write("{0,-20}", n);
      foreach(string v in resp.Headers.GetValues(n))
        Console.WriteLine(v);
    }

    // Close the response.
    resp.Close();
  }
}
```

Here is the output that was produced. (Remember, all header information is subject to change, so the precise output that you see may differ.)

```
Name                Value

Transfer-encoding   chunked
Content-Type        text/html
Date                Fri, 27 Jun 2008 20:32:06 GMT
Server              Sun-ONE-Web-Server/6.1
```

Accessing Cookies

You can gain access to the cookies associated with an HTTP response through the **Cookies** property defined by **HttpWebResponse**. Cookies contain information that is stored by a browser. They consist of name/value pairs, and they facilitate certain types of web access. The **Cookies** property is defined like this:

```
public CookieCollection Cookies { get; set; }
```

CookieCollection implements **ICollection** and **IEnumerable** and can be used like any other collection. (See Chapter 24.) It has an indexer that allows a cookie to be obtained by specifying its index or its name.

CookieCollection stores objects of type **Cookie**. **Cookie** defines several properties that give you access to the various pieces of information associated with a cookie. The two that we will use here are **Name** and **Value**, which are defined like this:

```
public string Name { get; set; }
public string Value { get; set; }
```

The name of the cookie is contained in **Name**, and its value is found in **Value**.

To obtain a list of the cookies associated with a response, you must supply a cookie container with the request. For this purpose, **HttpWebRequest** defines the property **CookieContainer**, shown here:

```
public CookieContainer CookieContainer { get; set; }
```

CookieContainer provides various fields, properties, and methods that let you store cookies. By default, this property is null. To use cookies, you must set it equal to an instance of the **CookieContainer** class. For many applications, you won't need to work with the **CookieContainer** property directly. Instead, you will use the **CookieCollection** obtained from the response. **CookieContainer** simply provides the underlying storage mechanism for the cookies.

The following program displays the names and values of the cookies associated with the URI specified on the command line. Remember, not all websites use cookies, so you might have to try a few until you find one that does.

```
/* Examine Cookies.

   To see what cookies a website uses,
   specify its name on the command line.
   For example, if you call this program
   CookieDemo, then

     CookieDemo http://msn.com

   displays the cookies associated with msn.com.
*/

using System;
using System.Net;

class CookieDemo {
  static void Main(string[] args) {

    if(args.Length != 1) {
      Console.WriteLine("Usage: CookieDemo <uri>");
      return ;
    }

    // Create a WebRequest to the specified URI.
    HttpWebRequest req = (HttpWebRequest)
           WebRequest.Create(args[0]);

    // Get an empty cookie container.
    req.CookieContainer = new CookieContainer();

    // Send the request and return the response.
    HttpWebResponse resp = (HttpWebResponse)
           req.GetResponse();

    // Display the cookies.
    Console.WriteLine("Number of cookies: " +
                      resp.Cookies.Count);
    Console.WriteLine("{0,-20}{1}", "Name", "Value");
```

```
    for(int i=0; i < resp.Cookies.Count; i++)
      Console.WriteLine("{0, -20}{1}",
                        resp.Cookies[i].Name,
                        resp.Cookies[i].Value);

    // Close the response.
    resp.Close();
  }
}
```

Using the LastModified Property

Sometimes you will want to know when a resource was last updated. This is easy to find out when using **HttpWebResponse** because it defines the **LastModified** property. It is shown here:

```
public DateTime LastModified { get; }
```

LastModified obtains the time that the content of the resource was last modified.

The following program displays the time and date at which the URI entered on the command line was last updated:

```
/* Use LastModified.

   To see the date on which a website was
   last modified, enter its URI on the command
   line. For example, if you call this program
   LastModifiedDemo, then to see the date of last
   modification for HerbSchildt.com enter

     LastModifiedDemo http://HerbSchildt.com
*/

using System;
using System.Net;

class LastModifiedDemo {
  static void Main(string[] args) {

    if(args.Length != 1) {
      Console.WriteLine("Usage: LastModifiedDemo <uri>");
      return ;
    }

    HttpWebRequest req = (HttpWebRequest)
           WebRequest.Create(args[0]);

    HttpWebResponse resp = (HttpWebResponse)
           req.GetResponse();

    Console.WriteLine("Last modified: " + resp.LastModified);

    resp.Close();
  }
}
```

MiniCrawler: A Case Study

To show how easy **WebRequest** and **WebReponse** make Internet programming, a skeletal web crawler called MiniCrawler will be developed. A *web crawler* is a program that moves from link to link to link. Search engines use web crawlers to catalog content. MiniCrawler is, of course, far less sophisticated than those used by search engines. It starts at the URI that you specify and then reads the content at that address, looking for a link. If a link is found, it then asks if you want to go to that link, search for another link on the existing page, or quit. Although this scheme is quite simple, it does provide an interesting example of accessing the Internet using C#.

MiniCrawler has several limitations. First, only absolute links that are specified using the **href="http** hypertext command are found. Relative links are not used. Second, there is no way to go back to an earlier link. Third, it displays only the links and no surrounding content. Despite these limitations, the skeleton is fully functional, and you will have no trouble enhancing MiniCrawler to perform other tasks. In fact, adding features to MiniCrawler is a good way to learn more about the networking classes and networking in general.

Here is the entire code for MiniCrawler:

```
/* MiniCrawler: A skeletal Web crawler.

   Usage:
     To start crawling, specify a starting
     URI on the command line. For example,
     to start at McGraw-Hill.com, use this
     command line:

       MiniCrawler http://McGraw-Hill.com

*/

using System;
using System.Net;
using System.IO;

class MiniCrawler {

  // Find a link in a content string.
  static string FindLink(string htmlstr,
                         ref int startloc) {
    int i;
    int start, end;
    string uri = null;
    string lowcasestr = htmlstr.ToLower();

    i = lowcasestr.IndexOf("href=\"http", startloc);
    if(i != -1) {
      start = htmlstr.IndexOf('"', i) + 1;
      end = htmlstr.IndexOf('"', start);
      uri = htmlstr.Substring(start, end-start);
      startloc = end;
    }
```

```
    return uri;
  }

  static void Main(string[] args) {
    string link = null;
    string str;
    string answer;

    int curloc; // holds current location in response

    if(args.Length != 1) {
      Console.WriteLine("Usage: MiniCrawler <uri>");
      return ;
    }

    string uristr = args[0]; // holds current URI

    try {

      do {
        Console.WriteLine("Linking to " + uristr);

        // Create a WebRequest to the specified URI.
        HttpWebRequest req = (HttpWebRequest)
               WebRequest.Create(uristr);

        uristr = null; // disallow further use of this URI

        // Send that request and return the response.
        HttpWebResponse resp = (HttpWebResponse)
               req.GetResponse();

        // From the response, obtain an input stream.
        Stream istrm = resp.GetResponseStream();

        // Wrap the input stream in a StreamReader.
        StreamReader rdr = new StreamReader(istrm);

        // Read in the entire page.
        str = rdr.ReadToEnd();

        curloc = 0;

        do {
          // Find the next URI to link to.
          link = FindLink(str, ref curloc);

          if(link != null) {
            Console.WriteLine("Link found: " + link);

            Console.Write("Link, More, Quit?");
            answer = Console.ReadLine();

            if(string.Compare(answer, "L", true) == 0) {
              uristr = string.Copy(link);
```

```
            break;
          } else if(string.Compare(answer, "Q", true) == 0) {
            break;
          } else if(string.Compare(answer, "M", true) == 0) {
            Console.WriteLine("Searching for another link.");
          }
        } else {
          Console.WriteLine("No link found.");
          break;
        }

      } while(link.Length > 0);

      // Close the response.
      resp.Close();
    } while(uristr != null);

  } catch(WebException exc) {
    Console.WriteLine("Network Error: " + exc.Message +
                      "\nStatus code: " + exc.Status);
  } catch(ProtocolViolationException exc) {
    Console.WriteLine("Protocol Error: " + exc.Message);
  } catch(UriFormatException exc) {
    Console.WriteLine("URI Format Error: " + exc.Message);
  } catch(NotSupportedException exc) {
    Console.WriteLine("Unknown Protocol: " + exc.Message);
  } catch(IOException exc) {
    Console.WriteLine("I/O Error: " + exc.Message);
  }

  Console.WriteLine("Terminating MiniCrawler.");
 }
}
```

Here is a short a sample session that begins crawling at McGraw-Hill.com. (Remember, the precise output will vary over time as content changes.)

```
Linking to http://mcgraw-hill.com
Link found: http://sti.mcgraw-hill.com:9000/cgi-bin/query?mss=search&pg=aq
Link, More, Quit? M
Searching for another link.
Link found: http://investor.mcgraw-hill.com/phoenix.zhtml?c=96562&p=irol-irhome
Link, More, Quit? L
Linking to http://investor.mcgraw-hill.com/phoenix.zhtml?c=96562&p=irol-irhome
Link found: http://www.mcgraw-hill.com/index.html
Link, More, Quit? L
Linking to http://www.mcgraw-hill.com/index.html
Link found: http://sti.mcgraw-hill.com:9000/cgi-bin/query?mss=search&pg=aq
Link, More, Quit? Q
Terminating MiniCrawler.
```

Let's take a close look at how MiniCrawler works. The URI at which MiniCrawler begins is specified on the command line. In **Main( )**, this URI is stored in the string called **uristr**. A request is created to this URI and then **uristr** is set to null, which indicates that this URI has already been used. Next, the request is sent and the response is obtained. The

content is then read by wrapping the stream returned by **GetResponseStream( )** inside a **StreamReader** and then calling **ReadToEnd( )**, which returns the entire contents of the stream as a string.

Using the content, the program then searches for a link. It does this by calling **FindLink( )**, which is a **static** method also defined by **MiniCrawler**. **FindLink( )** is called with the content string and the starting location at which to begin searching. The parameters that receive these values are **htmlstr** and **startloc**, respectively. Notice that **startloc** is a **ref** parameter. **FindLink( )** first creates a lowercase copy of the content string and then looks for a substring that matches **href="http**, which indicates a link. If a match is found, the URI is copied to **uri**, and the value of **startloc** is updated to the end of the link. Because **startloc** is a **ref** parameter, this causes its corresponding argument to be updated in **Main( )**, enabling the next search to begin where the previous one left off. Finally, **uri** is returned. Since **uri** was initialized to null, if no match is found, a null reference is returned, which indicates failure.

Back in **Main( )**, if the link returned by **FindLink( )** is not null, the link is displayed, and the user is asked what to do. The user can go to that link by pressing L, search the existing content for another link by pressing M, or quit the program by pressing Q. If the user presses L, the link is followed and the content of the link is obtained. The new content is then searched for a link. This process continues until all potential links are exhausted.

You might find it interesting to increase the power of MiniCrawler. For example, you might try adding the ability follow relative links. (This is not hard to do.) You might try completely automating the crawler by having it go to each link that it finds without user interaction. That is, starting at an initial page, have it go to the first link it finds. Then, in the new page, have it go to the first link and so on. Once a dead-end is reached, have it backtrack one level, find the next link, and then resume linking. To accomplish this scheme, you will need to use a stack to hold the URIs and the current location of the search within a URI. One way to do this is to use a **Stack** collection. As an extra challenge, try creating tree-like output that displays the links.

Using WebClient

Before concluding this chapter, a brief discussion of **WebClient** is warranted. As mentioned near the start of this chapter, if your application only needs to upload or download data to or from the Internet, then you can use **WebClient** instead of **WebRequest** and **WebResponse**. The advantage to **WebClient** is that it handles many of the details for you.

WebClient defines one constructor, shown here:

```
public WebClient( )
```

WebClient defines the properties shown in Table 25-6. **WebClient** defines a large number of methods that support both synchronous and asynchronous communication. Because asynchronous communication is beyond the scope of this chapter, only those methods that support synchronous requests are shown in Table 25-7. All methods throw a **WebException** if an error occurs during transmission.

The following program demonstrates how to use **WebClient** to download data into a file:

```
// Use WebClient to download information into a file.

using System;
using System.Net;
using System.IO;
```

```
class WebClientDemo {
  static void Main() {
    WebClient user = new WebClient();
    string uri = "http://www.McGraw-Hill.com";
    string fname = "data.txt";

    try {
      Console.WriteLine("Downloading data from " +
                        uri + " to " + fname);
      user.DownloadFile(uri, fname);
    } catch (WebException exc) {
      Console.WriteLine(exc);
    }

    Console.WriteLine("Download complete.");
  }
}
```


Property	Description
public string BaseAddress { get; set; }	Obtains or sets the base address of the desired URI. If this property is set, then addresses specified by the **WebClient** methods will be relative to the base address.
public RequestCachePolicy CachePolicy { get; set; }	Obtains or sets the policy that determines when the cache is used.
public ICredentials Credentials { get; set; }	Obtains or sets authentication information. This property is null by default.
public Encoding Encoding { get; set; }	Obtains or sets the character encoding used while transferring strings.
public WebHeaderCollection Headers{ get; set; }	Obtains or sets the collection of the request headers.
public bool IsBusy(get; }	If the request is still transferring information, this property is true. It is false otherwise.
public IWebProxy Proxy { get; set; }	Obtains or sets the proxy.
public NameValueCollection QueryString { get; set; }	Obtains or sets a query string consisting of name/value pairs that can be attached to a request. The query string is separated from the URI by a **?**. If more than one name/value pair exists, then an @ separates each pair.
public WebHeaderCollection ResponseHeaders{ get; }	Obtains a collection of the response headers.
public bool UseDefaultCredentials { get; set; }	Obtains or sets a value that determines if default credentials are used for authentication. If true, the default credentials (i.e., those of the user) are used. They are not used if false.

TABLE 25-6 The Properties Defined by **WebClient**

Method	Description
public byte[ ] DownloadData(string *uri*)	Downloads the information at the URI specified by *uri* and returns the result in an array of bytes.
public byte[ ] DownloadData(Uri *uri*)	Downloads the information at the URI specified by *uri* and returns the result in an array of bytes.
public void DownloadFile(string *uri*, string *fname*)	Downloads the information at the URI specified by *uri* and stores the result in the file specified by *fname*.
public void DownloadFile(Uri *uri*, string *fname*)	Downloads the information at the URI specified by *uri* and stores the result in the file specified by *fname*.
public string DownloadString(string *uri*)	Downloads the information at the URI specified by *uri* and returns the result as a **string**.
public string DownloadString(Uri *uri*)	Downloads the information at the URI specified by *uri* and returns the result as a **string**.
public Stream OpenRead(string *uri*)	Returns an input stream from which the information at the URI specified by *uri* can be read. This stream must be closed after reading is completed.
public Stream OpenRead(Uri *uri*)	Returns an input stream from which the information at the URI specified by *uri* can be read. This stream must be closed after reading is completed.
public Stream OpenWrite(string *uri*)	Returns an output stream to which information can be written to the URI specified by *uri*. This stream must be closed after writing is completed.
public Stream OpenWrite(Uri *uri*)	Returns an output stream to which information can be written to the URI specified by *uri*. This stream must be closed after writing is completed.
public Stream OpenWrite(string *uri*, string *how*)	Returns an output stream to which information can be written to the URI specified by *uri*. This stream must be closed after writing is completed. The string passed in *how* specifies how the information will be written.
public Stream OpenWrite(Uri *uri*, string *how*)	Returns an output stream to which information can be written to the URI specified by *uri*. This stream must be closed after writing is completed. The string passed in *how* specifies how the information will be written.
public byte[ ] UploadData(string *uri*, byte[ ] *info*)	Writes the information specified by *info* to the URI specified by *uri*. The response is returned.
public byte[ ] UploadData(Uri *uri*, byte[ ] *info*)	Writes the information specified by *info* to the URI specified by *uri*. The response is returned.
public byte[ ] UploadData(string *uri*, string *how*, byte[ ] *info*)	Writes the information specified by *info* to the URI specified by *uri*. The response is returned. The string passed in *how* specifies how the information will be written.

TABLE 25-7 The Synchronous Methods Defined by **WebClient**

This program downloads the information at McGrawHill.com and puts it into a file called **data.txt**. Notice how few lines of code are involved. By changing the string specified by **uri**, you can download information from any URI, including specific files.

Method	Description
public byte[ ] UploadData(Uri *uri*, string *how*, byte[ ] *info*)	Writes the information specified by *info* to the URI specified by *uri.* The response is returned. The string passed in *how* specifies how the information will be written.
public byte[ ] UploadFile(string *uri*, string *fname*)	Writes the information in the file specified by *fname* to the URI specified by *uri.* The response is returned.
public byte[ ] UploadFile(Uri *uri*, string *fname*)	Writes the information in the file specified by *fname* to the URI specified by *uri.* The response is returned.
public byte[ ] UploadFile(string *uri*, string *how*, string *fname*)	Writes the information in the file specified by *fname* to the URI specified by *uri.* The response is returned. The string passed in *how* specifies how the information will be written.
public byte[ ] UploadFile(Uri *uri*, string *how*, string *fname*)	Writes the information in the file specified by *fname* to the URI specified by *uri.* The response is returned. The string passed in *how* specifies how the information will be written.
public string UploadString(string *uri*, string *str*)	Writes *str* to the URI specified by *uri.* The response is returned.
public string UploadString(Uri *uri*, string *str*)	Writes *str* to the URI specified by *uri.* The response is returned.
public string UploadString(string *uri*, string *how*, string *str*)	Writes *str* to the URI specified by *uri.* The response is returned. The string passed in *how* specifies how the information will be written.
public string UploadString(Uri *uri*, string how, string *str*)	Writes *str* to the URI specified by *uri.* The response is returned. The string passed in *how* specifies how the information will be written.
public byte[ ] UploadValues(string *uri*, NameValueCollection *vals*)	Writes the values in the collection specified by *vals* to the URI specified by *uri.* The response is returned.
public byte[ ] UploadValues(Uri *uri*, NameValueCollection *vals*)	Writes the values in the collection specified by *vals* to the URI specified by *uri.* The response is returned.
public byte[ ] UploadValues(string *uri*, string *how*, NameValueCollection *vals*)	Writes the values in the collection specified by *vals* to the URI specified by *uri.* The response is returned. The string passed in *how* specifies how the information will be written.
public byte[ ] UploadValues(Uri *uri*, string *how*, NameValueCollection *vals*)	Writes the values in the collection specified by *vals* to the URI specified by *uri.* The response is returned. The string passed in *how* specifies how the information will be written.

TABLE 25-7 The Synchronous Methods Defined by **WebClient** *(continued)*

Although **WebRequest** and **WebResponse** give you greater control and access to more information, **WebClient** is all that many applications will need. It is particularly useful when all you need to do is download information from the Web. For example, you might use **WebClient** to allow an application to obtain documentation updates.

This page intentionally left blank

26
CHAPTER

Use System.Windows.Forms to Create Form-Based Windows Applications

Most of the programs shown in this book are console applications. Console applications are good for demonstrating the elements of the C# language and are appropriate for some types of utility programs, such as file filters. Of course, most modern applications are designed for the Windows' graphical user interface (GUI) environment, and this book would seem incomplete without demonstrating how to use C# to create a Windows application. Therefore, it is the topic of this, the final chapter in the book.

In the past, creating a Windows application was a challenging endeavor. It was not uncommon for a newcomer to spend several weeks just learning the basic elements and architecture of a Windows application. Fortunately, C# and the .NET Framework changes all that. The .NET Framework contains an entire subsystem devoted to Windows programming called *Windows Forms*. The primary support for Windows Forms is contained in the **System.Windows.Forms** namespace. Through the use of Windows Forms, the creation of a GUI-based Windows program has been greatly simplified, and the entire development process has been streamlined.

Before we begin it is important to emphasize one point: Windows programming is a *very* large topic, with entire series of books devoted to it. It is not possible to describe all aspects of it in a single chapter, nor is it possible to examine the classes, interfaces, properties, structures, and events in **System.Windows.Forms** in detail. There are just far too many. Instead, this chapter provides a "jump-start" to form-based Windows programming. It explains how to create a window, create a menu, implement a button, and respond to a message. Although these topics just scratch the surface of Windows programming, they will give you a base upon which to advance to other aspects of forms-based Windows programming.

A Brief History of Windows Programming

To appreciate the benefits that C# and the .NET Framework bring to Windows programming, it is necessary for you to understand a bit of its history. When Windows was first created, programs interacted directly with the Windows Application Programming Interface (API),

which is an extensive set of methods defined by Windows that programs call to access the various functions provided by Windows. API-based programs are very long and complicated. For example, even a skeletal API-based program requires about 50 lines of code. API-based programs that perform any useful function have *at least* several hundred lines of code, and real applications have several thousand lines of code. Thus, in the early days, Windows programs were difficult to write and maintain.

In response to this problem, class libraries were created that encapsulated the functionality of the API. The most important of these is the Microsoft Foundation Classes (MFC). Many readers of this book will be familiar with MFC. MFC is written in C++, and MFC-based programs are also written in C++. Because MFC brought object-oriented benefits, the process of creating a Windows program was simplified. However, MFC programs were still fairly complicated affairs, involving separate header files, code files, and resource files. Furthermore, MFC was only a "thin wrapper" around the API, so many Windows-based activities still required a significant number of explicit program statements.

C# and the .NET Framework's Forms library offer a fully object-oriented way to approach Windows programming. Instead of providing just a wrapper around the API, the Forms library defines a streamlined, integrated, logically consistent way of managing the development of a Windows application. This level of integration is made possible by the unique features of the C# language, such as delegates and events. Furthermore, because of C#'s use of garbage collection, the especially troubling problem of "memory leaks" has been nearly eliminated.

If you have already programmed for Windows using either the API or MFC, you will find the Windows Forms approach remarkably refreshing. Windows Forms makes it nearly as easy to create a Windows application as it is to create a console application.

Two Ways to Write a Form-Based Windows Application

Before we begin, an important point needs to be made. Visual Studio includes a sophisticated set of design tools that automate much of the process of creating a Windows application. Using these tools, you can visually construct and position the various controls and menus used by your application. Visual Studio will also "rough in" the classes and methods that are needed for each feature. Frankly, using the Visual Studio design tools are a good choice for creating most real-world Windows applications. However, there is no requirement that you use those tools. You can also create a Windows program by using a text editor and then compiling it, just like you can do for console-based applications.

Because this book is about C#, not Visual Studio, and because the Windows programs contained in this chapter are quite short, all programs will be shown in a form in which they can be entered using a text editor. However, the general structure, design, and organization of the programs is the same as that created by the design tools. Thus, the material in this chapter applies to either approach.

How Windows Interacts with the User

The first thing that you must learn about Windows programming is how the user and Windows interact because this defines the architecture that all Windows programs share. This interaction is fundamentally different from the console-based programs shown in the other parts of this book. When you write a console program, it is your program that initiates interaction with the operating system. For example, it is the program that requests such

things as input and output by calling **Read( )** or **WriteLine( )**. Thus, console programs call the operating system. The operating system does not call your program. However, in large measure, Windows works in the opposite way. It is Windows that calls your program. The process works like this: A program waits until it is sent a *message* by Windows. Once a message is received, your program is expected to take an appropriate action. Your program may call a method defined by Windows when responding to a message, but it is still Windows that initiates the activity. More than anything else, it is the message-based interaction with Windows that dictates the general form of all Windows programs.

There are many different types of messages that Windows may send to your program. For example, each time the mouse is clicked on a window belonging to your program, a mouse-clicked message will be sent. Another type of message is sent when a button is pressed or when a menu item is selected. Keep one fact firmly in mind: As far as your program is concerned, messages arrive randomly. This is why Windows programs resemble interrupt-driven programs. You can't know what message will be next.

Windows Forms

At the core of a C# Windows program is the *form*. A form encapsulates the basic functionality necessary to create a window, display it on the screen, and receive messages. A form can represent any type of window, including the main window of the application, a child window, or even a dialog box.

When a form is first created, it is empty. To supply functionality, you add menus and controls, such as pushbuttons, lists, and check boxes. Thus, you can think of a form as a container for other Windows objects.

When a message is sent to the window, it is translated into an event. Therefore, to handle a Windows message, you will simply register an event handler for that message with the form. Then, whenever that message is received, your event handler is automatically called.

The Form Class

A form is created by instantiating an object of the **Form** class or of any class derived from **Form**. **Form** contains significant functionality of its own, and it inherits additional functionality. Two of its most important base classes are **System.ComponentModel.Component**, which supports the .NET component model, and **System.Windows.Forms.Control**. The **Control** class defines features common to all Windows controls. Because **Form** inherits **Control**, it, too,
is a control. This fact allows forms to be used to create controls. Several of the members of **Form** and **Control** are used in the examples that follow.

A Skeletal Form-Based Windows Program

We will begin by creating a minimal form-based Windows application. This application simply creates and displays a window. It contains no other features. However, this skeleton does show the steps necessary to construct a fully functional window. This framework is the starting point upon which most types of Windows applications will be built. The skeletal form-based Windows program is shown next.

```
// A form-based Windows Skeleton.

using System;
using System.Windows.Forms;

// WinSkel is derived from Form.
class WinSkel : Form {

  public WinSkel() {
    // Give the window a name.
    Text = "A Windows Skeleton";
  }

  // Main is used only to start the application.
  [STAThread]
  static void Main() {
    WinSkel skel = new WinSkel(); // create a form

    // Enable visual styles.
    Application.EnableVisualStyles();

    // Start the window running.
    Application.Run(skel);
  }
}
```

The window created by this program is shown in Figure 26-1. Let's examine this program line-by-line. First, notice that both **System** and **System.Windows.Forms** are included. **System** is needed because of the **STAThread** attribute that precedes **Main( )**. **System.Windows.Forms** supports the Windows Forms subsystem, as just explained.

Next, a class called **WinSkel** is created. It inherits **Form**. Thus, **WinSkel** defines a specific type of form. In this case, it is a minimal form.

Inside the **WinSkel** constructor is the following line of code:

```
Text = "A Windows Skeleton";
```

Text is the property that sets the title of the window. Thus, this assignment causes the title bar in the window to contain **A Windows Skeleton**. **Text** is defined like this:

```
public override string Text { get; set; }
```

FIGURE 26-1 The skeletal form-based window

Text is inherited from **Control**.

Next is the **Main( )** method, which is declared much like the **Main( )** methods found throughout the rest of this book. It is the method at which program execution begins. Notice, however, that it is preceded by the **STAThread** attribute. As a general rule, the **Main( )** method of a Windows program should have this attribute. It sets the threading model for the program to a *single-threaded apartment* (STA). (A discussion of threading models and apartments is beyond the scope of this chapter, but briefly, a Windows application can use one of two different threading models: single-threaded apartment or multithreaded apartment.)

Inside **Main( )**, a **WinSkel** object called **skel** is created. Then, the **static** method **Application.EnableVisualStyles( )** is called. This method enables visual styles (themes) if the operating system supports it. It's not technically necessary in a Windows Forms application, but should be used because it enables better looking controls. It should be the first method called by **Main( )**. The **Application** class is defined within **System.Windows.Forms**, and it encapsulates aspects common to all Windows applications.

Next, **skel** is passed to the **Run( )** method, which is another method defined by the **Application** class, as shown here:

```
Application.Run(skel);
```

This starts the window running. The **Run( )** method used by the skeleton is shown here:

public static void Run(Form *form*)

It takes a reference to a form as a parameter. Since **WinSkel** inherits **Form**, an object of type **WinSkel** can be passed to **Run( )**.

When the program is run, it creates the window shown in Figure 26-1. The window has the default size and is fully functional. It can be resized, moved, minimized, maximized, and closed. Thus, the basic features needed by nearly all windows were achieved by writing only a few lines of form-based code. In contrast, the same program written using the C language and directly calling the Windows API would have required approximately five times as many lines of code!

The preceding skeleton defines the outline that most form-based Windows applications will take. In general, to create a form, you create a class that inherits **Form**. Initialize the form to meet your needs, create an object of your derived class, and then call **Application.Run( )** on that object.

Compiling the Windows Skeleton

You can compile a Windows program using either the command-line compiler or Visual Studio. For the very short programs shown in this chapter, the command-line compiler is the easiest way; but for real applications, you will probably want to use the IDE. (Also, as explained at the start of this chapter, you will probably want to use the design tools provided by Visual Studio, but that is not the approach described here.) Each way is shown here.

Compiling from the Command Line

Assuming that you call the skeleton **WinSkel.cs**, then to compile the skeleton from the command line, use this command:

```
csc /t:winexe WinSkel.cs
```

The **/t:winexe** switch tells the compiler to create a Windows application rather than a console program. To run the program, simply enter **WinSkel** at the command line.

Compiling from the IDE

To compile the program using the Visual Studio IDE, first create a new Windows Application project. Do this by selecting File | New Project. Then select Windows Forms Application in the New Project dialog box. Call the project **WinSkel**. Delete all C# source code files that were automatically created. Next, right-click on the WinSkel project name and select Add and then select New Item. From the Add New Item dialog box, select C# Code File, and name the file **MyWinProject.cs**. Now, enter the skeleton code exactly as shown, and then build the solution. To run the project, select Debug | Start Without Debugging.

Adding a Button

In general, the functionality of a window is expressed by two types of items: controls and menus. It is through these items that a user interacts with your program. Menus are described later in this chapter. Here you will see how to add a control to a window.

Windows defines many different types of controls, including pushbuttons, check boxes, radio buttons, and list boxes, to name just a few. Although each type of control is different, they all work in more or the less the same way. Here, we will add a pushbutton to a window, but the same basic procedure can be used to add other types of controls.

Button Basics

A pushbutton is encapsulated by the **Button** class. It inherits the abstract class **ButtonBase**, which inherits the **Control** class. **Button** defines only one constructor, which is shown here:

```
public Button( )
```

This creates a button that has a default size and location within the window. It contains no description. Before a button can be used, it will need to be given a description by assigning a string to its **Text** property.

To specify the location of the button within the window, you must assign the coordinates of its upper-left corner to the **Location** property. The **Location** property is inherited from **Control** and defined like this:

```
public Point Location { get; set; }
```

The coordinates are contained within a **Point** structure, which is defined in the **System.Drawing** namespace. It includes these two properties:

```
public int X { get; set; }
public int Y { get; set; }
```

Thus, to create a button that contains the description "Press Here" and is positioned at location 100, 200, use the following sequence:

```
Button MyButton = new Button();
MyButton.Text = "Press Here";
MyButton.Location = new Point(100, 200);
```

Adding a Button to a Form

After you have created a button, you must add it to a form. You do this by calling the **Add( )** method on the collection of controls linked to that form. This collection is available through

the **Controls** property, which is inherited from the **Control** class. The **Add( )** method is defined like this:

public virtual void Add(Control *cntl*)

Here, *cntl* is the control being added. Once a control has been added to a form, it will be displayed when the form is displayed.

A Simple Button Example

The following program adds a button to the skeleton shown earlier. At this time, the button does not do anything, but it is present in the form and can be clicked.

```
// Add a Button.

using System;
using System.Windows.Forms;
using System.Drawing;

class ButtonForm : Form {
  Button MyButton;

  public ButtonForm() {
    Text = "Using a Button";

    MyButton = new Button();
    MyButton.Text = "Press Here";
    MyButton.Location = new Point(100, 200);

    Controls.Add(MyButton);
  }

  [STAThread]
  static void Main() {
    ButtonForm skel = new ButtonForm();

    Application.EnableVisualStyles();
    Application.Run(skel);
  }
}
```

This program creates a class called **ButtonForm**, which is derived from **Form**. It contains a **Button** field called **MyButton**. Inside the constructor, the button is created, initialized, and added to the form. When run, the program displays the window shown in Figure 26-2. You can click the button, but nothing will happen. To make the button do something, you must add a message handler, as described in the next section.

Handling Messages

In order for a program to respond to a button press (or any other type of control interaction), it must handle the message that the button generates. In general, when a user interacts with a control, those interactions are passed to your program as messages. In a form-based C# program, these messages are processed by event handlers. Therefore, to receive messages,

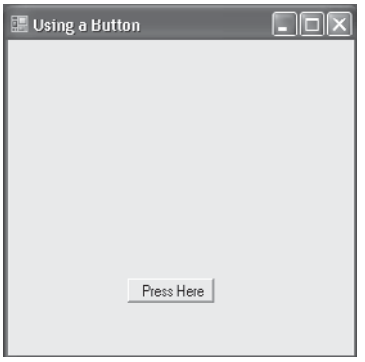


FIGURE 26-2
Adding a button

your program adds its own event handler onto the list of handlers called when a message is generated. For button-press messages, this means adding your handler to the **Click** event.

The **Click** event is specified by **Control** and defined by **Button**. It has this general form:

public Event EventHandler Click;

The **EventHandler** delegate is defined as shown here:

public delegate void EventHandler(object *who*, EventArgs *args*)

The object that generated the event is passed in *who*. Any information associated with that event is passed in *args*. For many events, *args* will be an object of a class derived from **EventArgs**. Since a button click does not require any additional information, we don't need to worry about event arguments when handling the button.

The following program adds button-response code to the preceding program. Each time the button is clicked, the location of the button is changed.

```
// Handle button messages.

using System;
using System.Windows.Forms;
using System.Drawing;

class ButtonForm : Form {
  Button MyButton;

  public ButtonForm() {
    Text = "Respond to a Button";

    MyButton = new Button();
    MyButton.Text = "Press Here";
    MyButton.Location = new Point(100, 200);

    // Add button event handler to list.
    MyButton.Click += MyButtonClick;

    Controls.Add(MyButton);
  }
```

```
  [STAThread]
  static void Main() {
    ButtonForm skel = new ButtonForm();

    Application.EnableVisualStyles();
    Application.Run(skel);
  }

  // Handler for MyButton.
  void MyButtonClick(object who, EventArgs e) {

    if(MyButton.Top == 200)
      MyButton.Location = new Point(10, 10);
    else
      MyButton.Location = new Point(100, 200);
  }
}
```

Let's look closely at the event-handling code in this program. The event handler for the button click is shown here:

```
// Handler for MyButton.
void MyButtonClick(object who, EventArgs e) {

  if(MyButton.Top == 200)
    MyButton.Location = new Point(10, 10);
  else
    MyButton.Location = new Point(100, 200);
}
```

MyButtonClick( ) is compatible with the **EventHandler** delegate shown earlier, which means it can be added to the **Click** event chain. Notice it is private to **ButtonForm**. This is not technically necessary, but it is a good idea because event handlers are not intended to be called except in response to events.

Inside the handler, the location of the top of the button is determined from the **Top** property. All controls define the following properties, which specify the coordinates of the upper-left and lower-right corners:

```
public int Top { get; set; }
public int Bottom { get; }
public int Left { get; set; }
public int Right { get; }
```

Notice that the location of the control can be changed by setting **Top** and **Left**, but not by setting **Bottom** and **Right** because they are read-only. (To change the size of a control, you can use the **Width** and **Height** properties.)

When the button click event is received, if the top of the control is at its original location of 200, the location is changed to 10, 10. Otherwise, it is returned to its original location of 100, 200. Therefore, each time you click the button, the location of the button changes.

Before **MyButtonClick( )** can receive messages, it must be added to the event handler chain linked to the button's **Click** event. This is done inside the **ButtonForm** constructor, using this statement:

```
MyButton.Click += MyButtonClick;
```

After the method is added to the **Click** event, each time the button is clicked, **MyButtonClick( )** is called.

An Alternative Implementation

As a point of interest, **MyButtonClick( )** could have been written in a slightly different way. Recall that the *who* parameter of an event handler receives a reference to the object that generated the call. In the case of a button click event, this is the button that was clicked. Thus, **MyButtonClick( )** could have been written like this:

```
// An Alternative button handler.
void MyButtonClick(object who, EventArgs e) {
  Button b = (Button) who;

  if(b.Top == 200)
    b.Location = new Point(10, 10);
  else
    b.Location = new Point(100, 200);
}
```

In this version, **who** is cast to **Button**, and this reference (rather than the **MyButton** field) is used to access the button object. Although there is no advantage to this approach in this case, it is easy to imagine situations in which it would be quite valuable. For example, such an approach allows a button event handler to be written independently of any specific button.

Using a Message Box

One of the most used built-in features of Windows is the *message box*. A message box is a predefined window that lets you display a message. You can also obtain simple responses from the user, such as Yes, No, or OK. In a form-based program, a message box is supported by the **MessageBox** class. You don't create an object of that class, however. Instead, to display a message box, call the **static** method **Show( )**, which is defined by **MessageBox**.

The **Show( )** method has several forms. The one we will be using is shown here:

public static DialogResult Show(string *msg*, string *caption*,
MessageBoxButtons *mbb*)

The string passed through *msg* is displayed in the body of the box. The caption of the message box window is passed in *caption*. The buttons that will be displayed are specified by *mbb*. The user's response is returned.

MessageBoxButtons is an enumeration that defines the following values:

AbortRetryIgnore	OK	OKCancel
RetryCancel	YesNo	YesNoCancel

Each of these values describes the buttons that will be included in a message box. For example, if *mbb* contains **YesNo**, then the Yes and No buttons are included in the message box.

The value returned by **Show( )** indicates which button was pressed. It will be one of these values, defined by the **DialogResult** enumeration:

Abort	Cancel	Ignore	No
None	OK	Retry	Yes

Your program can examine the return value to determine the course of action the user desires. For example, if the message box prompts the user before overwriting a file, your program can prevent the overwrite if the user clicks Cancel, and it can allow the overwrite if the user clicks OK.

The following program adds a stop button and a message box to the preceding example. In the stop button handler, a message box is displayed that asks the user if he or she wants to stop the program. If the user clicks Yes, the program is stopped. If the user clicks No, the program continues running.


```
// Add a stop button.

using System;
using System.Windows.Forms;
using System.Drawing;

class ButtonForm : Form {
  Button MyButton;
  Button StopButton;

  public ButtonForm() {
    Text = "Adding a Stop Button";

    // Create the buttons.
    MyButton = new Button();
    MyButton.Text = "Press Here";
    MyButton.Location = new Point(100, 200);

    StopButton = new Button();
    StopButton.Text = "Stop";
    StopButton.Location = new Point(100, 100);

    // Add the button event handlers to the window.
    MyButton.Click += MyButtonClick;
    Controls.Add(MyButton);
    StopButton.Click += StopButtonClick;
    Controls.Add(StopButton);
  }

  [STAThread]
  static void Main() {
    ButtonForm skel = new ButtonForm();

    Application.EnableVisualStyles();
    Application.Run(skel);
  }

  // Handler for MyButton.
  void MyButtonClick(object who, EventArgs e) {

    if(MyButton.Top == 200)
      MyButton.Location = new Point(10, 10);
    else
```

```
      MyButton.Location = new Point(100, 200);
  }

  // Handler for StopButton.
  void StopButtonClick(object who, EventArgs e) {

    // If user answers Yes, terminate the program.
    DialogResult result = MessageBox.Show("Stop Program?",
                            "Terminate",
                            MessageBoxButtons.YesNo);

    if(result == DialogResult.Yes) Application.Exit();
  }
}
```

Let's look closely at how the message box is used. Inside the **ButtonForm** constructor, a second button is added. This button contains the text "Stop," and its event handler is linked to **StopButtonClick( )**.

Inside **StopButtonClick( )**, the message box is displayed by the following statement:

```
// If user answers Yes, terminate the program.
DialogResult result = MessageBox.Show("Stop Program?",
                        "Terminate",
                        MessageBoxButtons.YesNo);
```

Here, the message inside the box is "Stop Program?", the caption is "Terminate," and the buttons to be displayed are Yes and No. When **Show( )** returns, the user's response is assigned to **result**. That response is then examined by the following statement to determine the course of action:

```
if(result == DialogResult.Yes) Application.Exit();
```

If the user clicks the Yes button, the program is stopped by calling **Application.Exit( )**, which causes the immediate termination of the program. Otherwise, no action is taken, and the program continues running.

Sample output is shown in Figure 26-3.

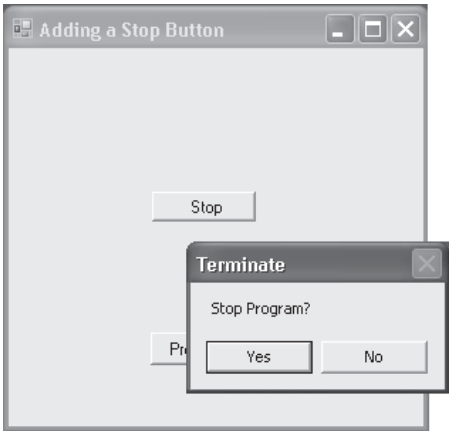


FIGURE 26-3 Sample output from the Stop Button program

Adding a Menu

The main window of nearly all Windows applications includes a menu across the top. This is called the *main menu.* The main menu typically contains top-level categories, such as File, Edit, and Tools. From the main menu descend *drop-down menus,* which contain the actual selections associated with the categories. When a menu item is selected, a message is generated. Therefore, to process a menu selection, your program will assign an event handler to each menu item. Because menus are such a bedrock resource in Windows programming, many options are available and the topic of menus is quite large. Fortunately, it is easy to create a simple main menu.

Originally, there was only one way to create a main menu: by using the classes **MainMenu** and **MenuItem**, both of which inherit the **Menu** class. These classes create the traditional-style menus that have been used in Windows applications for years. This approach is still supported. However, version 2.0 of the .NET Framework added a second way to add menus to a window by using a set of classes based on **ToolStrip**, such as **MenuStrip** and **ToolStripMenuItem**. Menus based on **ToolStrip** have many additional capabilities and give a modern look and feel. Understand, though, that both approaches to menus are currently valid for new C# applications.

Because there are two ways to create menus, both ways are described here, beginning with the traditional approach.

Creating a Traditional-Style Main Menu

A traditional-style main menu is constructed from a combination of two classes. The first is **MainMenu**, which encapsulates the overall structure of the menu. The second is **MenuItem**, which encapsulates an individual selection. A menu selection can either represent a final action, such as Close, or activate another drop-down menu. As mentioned, both **MainMenu** and **MenuItem** inherit the **Menu** class.

When a menu item is selected, a **Click** event is generated. **Click** is defined by **MenuItem**. Therefore, to handle a menu selection, your program will add its handler to the **Click** event list for that item.

Each form has a **Menu** property, which is defined like this:

```
public MainMenu Menu { get; set; }
```

By default, no menu is assigned to this property. To display a main menu, this property must be set to the menu that you create.

Creating a main menu is straightforward, but it does involve several steps. Here is the approach we will use:

1. Create a **MainMenu** object.
2. To the **MainMenu** object, add **MenuItem**s that describe the top-level categories. This is done by calling **Add( )** on the collection referred to by the **MenuItems** property.
3. To each top-level **MenuItem**, add the list of **MenuItem**s that defines the drop-down menu associated with that top-level entry. This is also done by calling **Add( )** on the collection referred to by the **MenuItems** property.
4. Add the event handlers for each selection.
5. Assign the **MainMenu** object to the **Menu** property associated with the form.

The following sequence shows how to create a File menu that contains three selections: Open, Close, and Exit:

```
// Create a main menu object.
MainMenu MyMenu  = new MainMenu();

// Add a top-level menu item to the menu.
MenuItem m1 = new MenuItem("File");
MyMenu.MenuItems.Add(m1);

// Create File submenu.
MenuItem item1 = new MenuItem("Open");
m1.MenuItems.Add(item1);

MenuItem item2 = new MenuItem("Close");
m1.MenuItems.Add(item2);

MenuItem item3 = new MenuItem("Exit");
m1.MenuItems.Add(item3);
```

Let's examine this sequence carefully. It begins by creating a **MainMenu** object called **MyMenu**. This object will be at the top of the menu structure. This object is instantiated by use of the default constructor for **MainMenu**. This creates an empty menu.

Next, a menu item called **m1** is created. This is the File heading. It is created by using this **MenuItem** constructor:

public MenuItem(string *caption*)

Here, *caption* specifies the text that is displayed by the item. In this case, it is "File." This menu item is then added directly to **MyMenu** and is a top-level selection.

Notice that when **m1** is added to **MyMenu**, it is done through the read-only **MenuItems** property. **MenuItems** is a collection of the menu items that form a menu. It is defined by **Menu**, which means it is part of both the **MainMenu** and **MenuItem** classes. It as shown here:

public Menu.MenuItemCollection MenuItems { get; }

To add a menu item to a menu, call **Add( )** on **MenuItems**, passing in a reference to the item to add, as the example shows.

Next, the drop-down menu associated with File is created. Notice that these menu items are added to the File menu item, **m1**. When one **MenuItem** is added to another, the added item becomes part of the drop-down menu associated with the item to which it is added. Thus, after the items **item1** through **item3** have been added to **m1**, selecting File will cause a drop-down menu containing Open, Close, and Exit to be displayed.

Once the menu has been constructed, the event handlers associated with each entry must be assigned. As explained, a user making a selection generates a **Click** event. Thus, the following sequence assigns the handlers for **item1** through **item3**.

```
// Add event handlers for the menu items.
item1.Click += MMOpenClick;
item2.Click += MMCloseClick;
item3.Click += MMExitClick;
```

Therefore, if the user selects Exit, **MMExitClick( )** is executed.

Finally, the **MainMenu** object must be assigned to the **Menu** property of the form, as shown here:

```
Menu = MyMenu;
```

After this assignment takes place, the menu will be displayed when the window is created, and selections will be sent to the proper handler.

The following program puts together all the pieces and demonstrates how to create a main menu and handle menu selections.

```
// Add a Main Menu.

using System;
using System.Windows.Forms;

class MenuForm : Form {
  MainMenu MyMenu;

  public MenuForm() {
    Text = "Adding a Main Menu";

    // Create a main menu object.
    MyMenu  = new MainMenu();

    // Add top-level menu items to the menu.
    MenuItem m1 = new MenuItem("File");
    MyMenu.MenuItems.Add(m1);

    MenuItem m2 = new MenuItem("Tools");
    MyMenu.MenuItems.Add(m2);

    // Create File submenu.
    MenuItem item1 = new MenuItem("Open");
    m1.MenuItems.Add(item1);

    MenuItem item2 = new MenuItem("Close");
    m1.MenuItems.Add(item2);

    MenuItem item3 = new MenuItem("Exit");
    m1.MenuItems.Add(item3);

    // Create Tools submenu.
    MenuItem item4 = new MenuItem("Coordinates");
    m2.MenuItems.Add(item4);

    MenuItem item5 = new MenuItem("Change Size");
    m2.MenuItems.Add(item5);

    MenuItem item6 = new MenuItem("Restore");
    m2.MenuItems.Add(item6);


    // Add event handlers for the menu items.
    item1.Click += MMOpenClick;
    item2.Click += MMCloseClick;
```

```
  item3.Click += MMExitClick;
  item4.Click += MMCoordClick;
  item5.Click += MMChangeClick;
  item6.Click += MMRestoreClick;

  // Assign the menu to the form.
  Menu = MyMenu;
}

[STAThread]
static void Main() {
  MenuForm skel = new MenuForm();

  Application.EnableVisualStyles();
  Application.Run(skel);
}

// Handler for main menu Coordinates selection.
void MMCoordClick(object who, EventArgs e) {
  // Create a string that contains the coordinates.
  string size =
    String.Format("{0}: {1}, {2}\n{3}: {4}, {5} ",
                  "Top, Left", Top, Left,
                  "Bottom, Right", Bottom, Right);

  // Display a message box.
  MessageBox.Show(size, "Window Coordinates",
                  MessageBoxButtons.OK);
}

// Handler for main menu Change selection.
void MMChangeClick(object who, EventArgs e) {
  Width = Height = 200;
}

// Handler for main menu Restore selection.
void MMRestoreClick(object who, EventArgs e) {
  Width = Height = 300;
}

// Handler for main menu Open selection.
void MMOpenClick(object who, EventArgs e) {

  MessageBox.Show("Inactive", "Inactive",
                  MessageBoxButtons.OK);
}

// Handler for main menu Close selection.
void MMCloseClick(object who, EventArgs e) {

  MessageBox.Show("Inactive", "Inactive",
                  MessageBoxButtons.OK);
}

// Handler for main menu Exit selection.
```

```
  void MMExitClick(object who, EventArgs e) {

    DialogResult result = MessageBox.Show("Stop Program?",
                            "Terminate",
                             MessageBoxButtons.YesNo);

    if(result == DialogResult.Yes) Application.Exit();
  }
}
```

Sample output is shown in Figure 26-4.

This program defines two drop-down menus. The first is accessed via the File menu. It contains the Open, Close, and Exit selections. The menu handlers for Open and Close are simply placeholders that perform no function other than displaying a message box to that effect. The Close handler asks if you want to stop the program. If you answer Yes, the program is terminated.

The Tools menu has these selections: Coordinates, Change Size, and Restore. Selecting Coordinates causes the coordinates of the upper-left and lower-right corners of the window to be displayed in a message box. Try moving the window and then displaying its coordinates. Each time the window is moved to a new location, its coordinates change.

Choosing Change Size causes the window to be reduced in size so its width and height are both 200 pixels long. This is done through the **Width** and **Height** properties, shown here:

```
public int Width { get; set; }
public int Height { get; set; }
```

Selecting Restore returns the window to its default size.

Creating a New-Style Menu with MenuStrip

Although the traditional approach to creating menus described in the preceding section is still valid, the .NET Framework offers what is often a better way to manage menus that uses the **MenuStrip** control. **MenuStrip** gives your menus a modern look and additional capabilities, such as menu merging. Because of the advantages of using **MenuStrip**, its use is recommended.

The **MenuStrip** class is the modern equivalent of **MainMenu** described in the previous section. It is based on the **ToolStrip** class, which defines much of the functionality supported by the new approach to menus. **ToolStrip** is, essentially, a container for toolbar objects,

FIGURE 26-4 Sample output from the Menu program

including menu items. **MenuStrip** inherits this functionality and provides specific support for menus, including the main menu. Items that are stored in a **MenuStrip** are objects of type **ToolStripMenuItem**, which is the modern equivalent of **MenuItem**, also described in the previous section.

Using **MenuStrip** and **ToolStripMenuItem** is similar to using a **MainMenu** and **MenuItem** as described in the previous section. Here is one way to create and use a **MenuStrip**:

1. Create a **MenuStrip** control.
2. To the **MenuStrip** control, add **ToolStripMenuItem**s that describe the top-level categories. This is done by calling **Add( )** on the collection referred to by the **Items** property provided by **MenuStrip**.
3. To each top-level **ToolStripMenuItem**, add the list of **ToolStripMenuItem**s that defines the drop-down menu associated with that top-level entry. This is done by calling **Add( )** on the collection referred to by the **DropDownItems** property.
4. Add the event handlers for each selection.
5. Add the **MenuStrip** to the list of controls for the form by calling **Add( )** on the **Controls** property.
6. Assign the **MenuStrip** control to the **MainMenuStrip** property associated with the form.

Pay special attention to Steps 2 and 3 because they differ from the parallel steps used with **MainMenu**. First, in Step 2, top-level menu items that represent submenus are added to the collection referred to by **Items**. Second, in Step 3, individual menu items are added to the collection referred to by the **DropDownItems** property. (Both **MainMenu** and **MenuItem** use the **MenuItems** property for this purpose.) Other than those differences, the two procedures are essentially the same.

The following program demonstrates **MenuStrip** by reworking the program shown in the previous section. Sample output is shown in Figure 26-5. Notice that the menu now has a modern look.

```
// Use a MenuStrip.

using System;
using System.Windows.Forms;

class MenuForm : Form {
  MenuStrip MyMenu; // use a MenuStrip

  public MenuForm() {
    Text = "Use a MenuStrip";

    // Create a main menu object.
    MyMenu  = new MenuStrip();

    // Add top-level menu items to the menu.
    ToolStripMenuItem m1 = new ToolStripMenuItem("File");
    MyMenu.Items.Add(m1);

    ToolStripMenuItem m2 = new ToolStripMenuItem("Tools");
    MyMenu.Items.Add(m2);
```

```
  // Create File submenu.
  ToolStripMenuItem item1 = new ToolStripMenuItem("Open");
  m1.DropDownItems.Add(item1);

  ToolStripMenuItem item2 = new ToolStripMenuItem("Close");
  m1.DropDownItems.Add(item2);

  ToolStripMenuItem item3 = new ToolStripMenuItem("Exit");
  m1.DropDownItems.Add(item3);

  // Create Tools submenu.
  ToolStripMenuItem item4 = new ToolStripMenuItem("Coordinates");
  m2.DropDownItems.Add(item4);

  ToolStripMenuItem item5 = new ToolStripMenuItem("Change Size");
  m2.DropDownItems.Add(item5);

  ToolStripMenuItem item6 = new ToolStripMenuItem("Restore");
  m2.DropDownItems.Add(item6);


  // Add event handlers for the menu items.
  item1.Click += MMOpenClick;
  item2.Click += MMCloseClick;
  item3.Click += MMExitClick;
  item4.Click += MMCoordClick;
  item5.Click += MMChangeClick;
  item6.Click += MMRestoreClick;

  // Add to list of controls.
  Controls.Add(MyMenu);

  // Assign the menu to the form.
  MainMenuStrip = MyMenu;
}

[STAThread]
static void Main() {
  MenuForm skel = new MenuForm();

  Application.EnableVisualStyles();
  Application.Run(skel);
}

// Handler for main menu Coordinates selection.
void MMCoordClick(object who, EventArgs e) {
  // Create a string that contains the coordinates.
  string size =
    String.Format("{0}: {1}, {2}\n{3}: {4}, {5} ",
                  "Top, Left", Top, Left,
                  "Bottom, Right", Bottom, Right);

  // Display a message box.
  MessageBox.Show(size, "Window Coordinates",
```

```
                    MessageBoxButtons.OK);
  }

  // Handler for main menu Change selection.
  void MMChangeClick(object who, EventArgs e) {
    Width = Height = 200;
  }

  // Handler for main menu Restore selection.
  void MMRestoreClick(object who, EventArgs e) {
    Width = Height = 300;
  }

  // Handler for main menu Open selection.
  void MMOpenClick(object who, EventArgs e) {

    MessageBox.Show("Inactive", "Inactive",
                    MessageBoxButtons.OK);
  }

  // Handler for main menu Close selection.
  void MMCloseClick(object who, EventArgs e) {

    MessageBox.Show("Inactive", "Inactive",
                    MessageBoxButtons.OK);
  }

  // Handler for main menu Exit selection.
  void MMExitClick(object who, EventArgs e) {

    DialogResult result = MessageBox.Show("Stop Program?",
                            "Terminate",
                             MessageBoxButtons.YesNo);

    if(result == DialogResult.Yes) Application.Exit();
  }
}
```

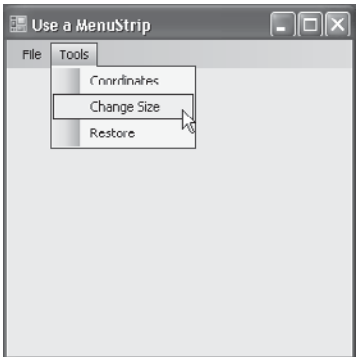


FIGURE 26-5 Sample output from the **MenuStrip** program

APPENDIX A

Documentation Comment Quick Reference

C# supports three types of comments. The first two are **//** and **/* */**. The third type is based on XML tags and is called a *documentation comment*. (The term *XML comment* is also commonly used.) A single-line documentation comment begins with **///**. A multiline documentation comment begins with **/**** and ends with ***/**. The lines after the **/**** can (but are not required to) start with a single *. If all subsequent lines in the multiline comment begin with a *, the * is ignored.

Documentation comments precede the declaration of such things as classes, namespaces, methods, properties, and events. Using documentation comments, you can embed information about your program into the program itself. When you compile the program, you can have the documentation comments placed into an XML file. Documentation comments can also be utilized by the IntelliSense feature of Visual Studio.

The XML Comment Tags

C# supports the XML documentation tags shown in Table A-1. Most of the XML comment tags are readily understandable, and they work like all other XML tags with which most programmers are already familiar. However, the **<list>** tag is more complicated than the others. A list contains two components: a list header and list items. The general form of a list header is shown here:

```
<listheader>
  <term> name </term>
  <description> text </description>
</listheader>
```

Here, *text* describes *name.* For a table, *text* is not used. The general form of a list item is shown next:

```
<item>
  <term> item-name </term>
  <description> text </description>
</item>
```

Here, *text* describes *item-name.* For bulleted or numbered lists or tables, *item-name* is not used. There can be multiple **<item>** entries.

Tag	Description
<c> *code* </c>	Specifies the text specified by *code* as program code.
<code> *code* </code>	Specifies multiple lines of text specified by *code* as program code.
<example> *explanation* </example>	The text associated with e*xplanation* describes a code example.
<exception cref = "*name*"> *explanation* </exception>	Describes an exception. The exception is specified by *name.*
<include file = *'fname'* path = *'path* [@*tagName = "tagID* "]' />	Specifies a file that contains the XML comments for the current file. The file is specified by *fname.* The path to the tag, the tag name, and the tag ID are specified by *path, tagName,* and *tagID,* respectively.
<list type = "*type*""> *list-header* *list-items* </list>	Specifies a list. The type of the list is specified by *type,* which must be either bullet, number, or table.
<para> *text* </para>	Specifies a paragraph of text within another tag.
<param name = *'param-name'*> *explanation* </param>	Documents the parameter specified by *param-name*. The text associated with e*xplanation* describes the parameter.
<paramref name = "*param-name*" />	Specifies that *param-name* is a parameter name.
<permission cref = "*identifier*"> *explanation* </permission>	Describes the permission setting associated with the class members specified by *identifier.* The text associated with e*xplanation* describes the permission settings.
<remarks> *explanation* </remarks>	The text specified by *explanation* is a general commentary often used to describe a type, such as a class or structure.
<returns> *explanation* </returns>	The text specified by explanation documents the return value of a method.
<see cref = "*identifier*" />	Declares a link to another element specified by *identifier.*
<seealso cref = "*identifier*" />	Declares a "see also" link to *identifier.*
<summary> *explanation* </summary>	The text specified by *explanation* is a general commentary often used to describe a method or other class member.
<typeparam name = "*param-name*"> *explanation* </typeparam>	Documents the type parameter specified by *param-name.* The text associated with e*xplanation* describes the type parameter.
<typeparamref name = "*param-name*"/>	Specifies that *param-name* is the name of a type parameter.
<value> *explanation* </value>	The text specified by *explanation* documents a property.

TABLE A-1 The XML Comment Tags

Compiling Documentation Comments

To produce an XML file that contains the documentation comments, specify the **/doc** option. For example, to compile a file called **DocTest.cs** that contains XML comments, use this command line:

```
csc DocTest.cs /doc:DocTest.xml
```

To create an XML output file when using the Visual Studio 2008 IDE, you must activate the Properties page. Next, select Build. Then, check the XML Documentation File box and specify the name of the XML file.

An XML Documentation Example

Here is an example that demonstrates several documentation comments. It uses both the multiline and the single-line forms. As a point of interest, many programmers use a series of single-line documentation comments rather than a multiline comment even when a comment spans several lines. (Several of the comments in this example use this approach.) The advantage is that it clearly identifies each line in a longer documentation comment as being part of a documentation comment. This is, of course, a stylistic issue, but it is common practice.


```
// A documentation comment example.

using System;

/** <remark>
 This is an example of multiline XML documentation.
 The Test class demonstrates several tags.
</remark>
*/

class Test {
  /// <summary>
  /// Main is where execution begins.
  /// </summary>
  static void Main() {
    int sum;

    sum = Summation(5);
    Console.WriteLine("Summation of " + 5 + " is " + sum);
  }

  /// <summary>
  /// Summation returns the summation of its argument.
  /// <param name = "val">
  /// The value to be summed is passed in val.
  /// </param>
  /// <see cref="int"> </see>
  /// <returns>
  /// The summation is returned as an int value.
  /// </returns>
  /// </summary>
  static int Summation(int val) {
    int result = 0;

    for(int i=1; i <= val; i++)
      result += i;

    return result;
  }
}
```

Assuming the preceding program is called **XmlTest.cs**, the following line will compile the program and produce a file called **XmlTest.xml** that contains the comments:

```
csc XmlTest.cs /doc:XmlTest.xml
```

After compiling, the following XML file is produced:

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>DocTest</name>
    </assembly>
    <members>
        <member name="T:Test">
            <remark>
             This is an example of multiline XML documentation.
             The Test class demonstrates several tags.
            </remark>
        </member>
        <member name="M:Test.Main">
            <summary>
            Main is where execution begins.
            </summary>
        </member>
        <member name="M:Test.Summation(System.Int32)">
            <summary>
            Summation returns the summation of its argument.
            <param name="val">
            The value to be summed is passed in val.
            </param>
            <see cref="T:System.Int32"> </see>
            <returns>
            The summation is returned as an int value.
            </returns>
            </summary>
        </member>
    </members>
</doc>
```

Notice that each documented element is given a unique identifier. These identifiers can be used by other programs that use the XML documentation.

Index

Symbols

&
  bitwise AND, 73–76
  logical AND, 67, 68, 70, 71, 229–235
  pointer operator, 587
&& (short-circuit or conditional AND), 67, 70–71, 229, 231–235
*
  multiplication operator, 24, 63
  pointer operator, 586–587, 594, 595
@ to designate verbatim string literals, 48–49
@ - qualified keywords, 33–34
|
  bitwise OR, 73, 76–77
  logical OR, 67, 68, 69–70, 71, 229–235
|| (short-circuit or conditional OR), 67, 70, 71, 229, 231–235
[ ], 137, 138, 142, 145, 151, 245, 483, 484, 598
^
  bitwise exclusive OR, 73, 77–78
  logical exclusive OR, 67, 68
: (colon), 271, 278, 334
:: namespace alias qualifier, 447–451
{ }, 20, 21, 29, 31, 52, 118, 140, 143, 414, 416, 569
  used with format specifiers, 43–46
=, 24, 71–72
= = (relational operator), 27, 67, 537
  overloading, 224, 226, 306
  and strings, 158, 665
=> lambda operator, 413–414
!, 67, 68, 69–70, 229–231
!=, 27, 67, 537
  overloading, 226
  and strings, 158, 665
/, 24, 63
/* */, 20
/** */, 867
//, 21
///, 867
<
  I/O redirection operator, 381–382
  relational operator, 27, 67, 224–226
< >, 497
<<, 73, 79–81
<=, 27, 67, 226
–, 24, 63
– =
  compound assignment operator, 72
  and delegates, 404
  and event accessors, 425
–>, 589
– –, 29, 63, 64–65
  overloading, 217
%
  modulus operator, 63–64
  used with custom date and time placeholder characters, 698
( ), 21, 83, 116, 122
. (dot operator), 111, 119, 205, 332, 589
+
  addition operator, 24, 63
  concatenating strings with, 159, 670
  used with WriteLine( ), 24
+=
  compound assignment operator, 72
  and delegates, 404
  and event accessors, 425
++, 29, 63, 64–67
  overloading, 217–220
#
  used with preprocessor directive, 451
  used with WriteLine( ), 45, 692–695
?
  conditional operator, 82–83
  used with nullable type declaration, 599
?? null coalescing operator, 601–602
>
  I/O redirection operator, 381–382
  relational operator, 27, 67, 224–226
>>, 73, 79–81
>=, 27, 67, 226
; (semicolon), 21, 31, 92, 143, 409, 548
  used in a custom format specifier, 693–694, 695
~
  bitwise NOT, 73, 78–79
  used in destructor declaration, 131–132

A

Abort( ), 741–745
Abs( ), 193, 618
abstract type modifier, 294, 301, 305
Access control, 165–172
  guidelines for using public and private, 167–168
Access modifiers/specifiers, 110, 116, 165–166, 261–264, 458
Accessors, 245
  event, 424–429
  indexer, 246, 248–249
  property, 254–255, 259–260
  using access modifiers with, 261–264
Acos( ), 618
Action delegate, 653–654
add event accessor, 425, 428
Add( ), 631, 752, 753, 763, 776, 777, 787, 790, 792, 794, 796, 852–853, 859, 860, 864
  and collection initializers, 819
AddAfter( ), 784
AddBefore( ), 784
AddFirst( ), 784
AddLast( ), 784
AddMemoryPressure( ), 658
AddRange( ), 756, 780
ADO.NET Datasets, 546
All( ) extension method, 578–579
AllKeys property, 834
AllowMultiple named parameter, 491
AND operator
  bitwise (&), 73–76
  logical (&), 67, 68, 70, 71, 229–235
  short-circuit or conditional (&&), 67, 70–71, 229, 231–235
And( ), 772
Any( ) extension method, 578–579
Application class, 851
Application.EnableVisualStyles( ), 851
Application.Exit( ), 858
ApplicationException, 354
ArgumentException, 372, 378, 380, 381, 384, 802, 805
ArgumentNullException, 372, 378, 380, 381, 384, 397, 753, 777, 830
ArgumentOutOfRangeException, 372, 758
Arguments, 21, 116, 122
  command-line, 200–202

Arguments *(continued)*
passing, 174–176
type. *See* Type argument(s)
variable number of, using, 182–185
Arithmetic operators, 24, 63–67
Array class, 641–654, 762
methods, table of, 642–647
properties, table of, 641–642
Array.ForEach(). *See* ForEach()
Array(s), 137–152
array of, 144, 146
boundaries, 141, 338
boundary errors, using indexer to prevent runtime exceptions caused by, 246–249, 252–254
boundary errors, using a property to prevent runtime exceptions caused by, 256–259, 262–264
copying an, 651–652
custom, using indexers and properties to create, 264–268
dynamic, 268, 755, 779
and fixed-size buffers, 596–598
foreach to cycle through, using, 153–156
general forms of, 137–138, 141, 142
and the IEnumerable<T> interface, 546, 547, 550, 566, 576
implemented in C# as objects, 137–138, 148
implicitly typed, 151–152
indexing (accessing elements of), 138–139, 142, 145
initializing, 140, 143–144
I/O using, 392–394
jagged, 144–146, 150–151, 152
Length property of, 148–151
and LINQ, 151, 152, 546
multidimensional, 141–146
null-terminated, 593
one-dimensional, 137–141
parameter params, 182–185
of pointers, 595
pointers and, 591–593
rectangular, 144
reference variables, assigning, 146–147
returned from methods, 187–188
reversing an, 650–651
sorting, 648–650
searching, 155–156, 648–650
of strings, 160–161
ArrayList class, 749, 755–762, 774, 779
methods, table of, 756–758
ArrayTypeMismatchException, 353
Arrow operator (–>), 589
as operator, 459, 460–462
ascending contextual keyword, 550, 553
ASCII character set, 41, 363
ASCIIEncoding, 664
Asin(), 618
ASP.NET, 457, 821
AsQueryable() extension method, 752
AsReadOnly<T>(), 642
Assembly, 166, 457–458
alias, 610–611
and DLL files, 480–481
friend, 605
and reflection, 475–481
Assembly class, 476
Assembly property, 464
Assignment operator(s)
=, 24, 71–72
arithmetic and logical compound (*op*=), 72
bitwise compound, 81
and operator overloading, 239–240
Assignments
reference variables and, 115–116, 146–147, 328, 329–330
structure, 328–329
type conversion in, 55–58
Atan(), 618
Atan2(), 618
Attribute class, 483, 484, 485
Attributes, 459, 464, 483–493
attaching, 484–485
built-in, 491–493
creating, 484
and named parameters, 487–490
retrieving, 485–486
Attributes property, 464
AttributeTargets enumeration, 491
AttributeUsage attribute, 484, 491
AuthenticationLevel property, 825
AutoResetEvent class, 737, 739
Average() extension method, 578–579

B

Backing field, 259, 260
Backslash character constants, 47
base keyword
to access a hidden base class member, 278, 283–285
to call a base class constructor, 278–282, 288, 355, 358
BaseAddress property, 843
BaseType property, 464
BigMul(), 618
BinaryReader class, 367, 383, 384–390
input methods, table of common, 385
BinarySearch(), 642, 648, 756, 760–761, 780
BinaryWriter class, 367, 383–384, 386–390
output methods, table of common, 384
BindingFlags enumeration, 468
Bit-fields, 700
BitArray class, 750, 771–774
methods, table of, 772
BitConverter class, 654–656
methods, table of, 655–656
Bitwise operators, 73–81
Blocks, code, 21, 29–31, 52
and anonymous methods, 409–411
and name hiding, 54
bool value type, 36, 42–43, 623
and logical operators, 67, 68
and relational operators, 43, 67
Boolean expression, 27
Boolean structure, 623, 640–641
methods, table of, 640–641
Bottom property, 855
Boxing and unboxing, 307–309
break statement, 85, 89, 91, 102–104, 107, 154, 815
BufferedStream class, 365
Buffers, fixed-size, 596–598
Button class, 852, 854, 856
ButtonBase class, 852
Byte .NET structure, 396, 397, 623
byte value type, 36, 37, 38, 363, 396, 623
Bytecode, 5, 8

C

C format specifier, 45–46
C, history of, 3–4
C++, history of, 4
C#
as a component-oriented language, 6–7
case sensitivity of, 21
history of, 3, 5–7
Internet and, 821
and Java, 3, 6

keywords, 32–33
library, 8, 20, 34, 38, 437, 613, 615
and the .NET Framework, 3, 6, 7–8
program compilation, 8, 14–19
as a strongly typed language, 35, 289
version 3.0, new features in, 7
CachePolicy property, 825, 843
Call-by-reference
using ref to create a, 176–178
vs. call-by-value, 174–176
CanRead property, 365
CanSeek property, 365
CanTimeout property, 365
CanWrite property, 365
Capacity property, 758, 781, 793
Case sensitivity and C#, 21
case statement, 89–92
and goto, 106–107
Cast( ) extension method, 752
Cast(s), 42, 55, 56–58, 60–62, 496, 499, 501, 502
as operator, using the, 460–462
and enumerations, 332–333
and explicit conversion operators, 235, 237–239, 240
catch clause, 338–345
for catching all exceptions, 345–346
and catching derived class exceptions, 358–359
and finally, 351
multiple, using, 344–345
variable, 338, 340, 351
Ceiling( ), 618, 631
Char structure, 623, 634–640
methods, table of, 635–638
char value type, 36, 41–42, 363, 623
Character(s), 41–42
escape sequences, 47
literals, 46, 47, 49
CharacterSet property, 835
checked keyword, 360–362
checksum #pragma option, 457
Class(es), 109–114
abstract, 302–304, 305, 311, 326
base, definition of, 269
constructor. *See* Constructor(s)
data type, definition creating a new, 111
definition of the term, 8, 12, 20, 109
derived, definition of, 269
factory, 186–187, 209–210
general form of, 110, 272
generic. *See* Generic class
and interfaces, 311, 312–316
library, .NET, 8, 20, 34, 38, 437, 613, 615
member. *See* Member(s), class
partial, 603–604
sealed, 305
static, 211–212
System, list of, 615–616
well-designed, 110
class keyword, 20, 110, 504, 517, 518, 521
Clear( ), 643, 752, 753, 767, 770, 776, 794, 795
Click event, 854, 859, 860
Clone( ), 643, 660–662
Close( ), 364, 366, 374, 376, 384, 385, 748, 827, 830
CloseMainWindow( ), 748
CLR (Common Language Runtime), 8, 9, 15, 337, 352, 585
CLS (Common Language Specification), 9
Code
blocks. *See* Blocks, code
managed vs. unmanaged, 9, 585, 586
point, Unicode character, 634–635
section, critical, 606
unreachable, 126
unsafe, 585–598
Collect( ), 657, 658
Collection(s), 326, 749–819
bit-based, 749, 750, 771–774
definition of, 102, 152, 749
initializers, 819
specialized, 749, 750, 774
sorting user-defined classes in, 801–808
storing user-defined classes in, 799–801
using an enumerator to access, 808–811
Collection(s), generic, 749, 750, 774–799
classes, 779–799
interfaces, 775–778
Collection(s), non-generic, 749–771
classes, 755–771
interfaces, 751–755
CollectionBase class, 774
CollectionCount( ), 658
Collections API, 749, 750, 774
CollectionsUtil class, 774
Comments, 20, 21, 867
Common Language Runtime (CLR), 8, 9, 15, 337, 352, 585
Common Language Specification (CLS), 9
Common Type System (CTS), 9
Compare( ), 631, 641, 665, 666–670, 754, 778, 805, 806–807
comparer property, 763
Comparer property, 788, 790, 793, 797
CompareOrdinal( ), 667
CompareTo( ), 157–158, 326, 534, 536–537, 624, 626, 628, 631, 635, 640, 648–649, 659, 667, 802
CompareTo(T obj), 536–537, 649, 660, 803–804
Compilation, conditional, 452
Compile( ), 581
Compiler
C# command-line, 14–15
JIT, 8, 15, 543
Components, 5
and C#, 6–7
and the internal access modifier, 458
Concat( ), 670–672
Conditional built-in attribute, 491–492
ConnectionGroupName property, 825
Console class, 21, 364, 367, 369, 371, 382
Console.Error, 363, 364, 367, 371
Console.In, 363, 364, 367–371, 381
Console.Out, 363–364, 367, 371, 381
ConsoleKey enumeration, 370
ConsoleKeyInfo structure, 369
ConsoleModifiers enumeration, 370
const modifier, 607
Constants, 35
backslash character, 47
named integer, 311, 332
ConstrainedCopy( ), 643
Constraint(s), 503–521
base class, 504–512, 520, 521
constructor, 504, 516–517, 521
and generic interfaces, 534
and generic methods, 527
interface, 504, 512–516, 521
multiple, using, 521
naked type, 504, 520
and overload resolution, 543
reference type, 504, 517, 518–519, 521
value type, 504, 517, 519
Constructed type, 499, 543–544
and method overloading, 543
open vs. closed, 499
and static fields, 544

Constructor(s), 126–130
base clause to call a base class, 278–282, 288
in class hierarchy, order of calling, 288–289
default, 127, 130–131
and inheritance, 276–282, 288, 291–294
object initializers vs., 199–200
overloading, 194–199, 543
parameterless, 504, 516–517
reflection to obtain a type's, using, 471–475
static, 210, 212
ConstructorInfo class, 472
Contains( ), 674–675, 752, 753, 767, 770, 776, 794, 795
extension method, 578–579
ContainsKey( ), 762, 763, 764, 765, 777, 787, 790, 792
ContainsValue( ), 762, 763, 764, 765, 787, 790, 792
ContentEncoding property, 835
ContentLength property, 825, 827, 835
ContentType property, 825, 827, 835
continue statement, 85, 104–105
Contravariance, 406–408
Control class, 849, 851, 852, 853, 854
Control statements. *See* Statements, control
Controls property, 853, 864
Controls, Windows, 849, 852, 855
Conversion operators, 235–239, 243–244
implicit vs. explicit, 235, 237, 239
Conversion, type. *See* Type conversion
ConvertAll<TFrom, TTo>( ), 643
ConvertFromUtf32( ), 634, 635
ConvertToUtf32( ), 634, 635
Cookie class, 836
CookieCollection, 836, 837
CookieContainer class, 837
CookieContainer property, 837
Cookies, accessing, 836–838
Cookies property, 835, 836
Copy( ), 157, 643, 651
CopyTo( ), 643, 751, 756–757, 776, 794, 795
Cos( ), 39, 618
Cosh( ), 618
Count property, 751, 758, 773, 775
Count( ) extension method, 564, 578–579, 580
Covariance, 406–408
Create( ), 824, 825, 829, 830, 833
exceptions, 830
and the Uri class, 833
CreateInstance( ), 643
Credentials property, 825, 843
Cross-language interoperability, 5, 457
.cs file extension, 20
csc.exe command-line compiler, 14–15
CTS (Common Type System), 9
CultureInfo class, 638, 666, 667
Current property, 754, 778, 808, 809, 810, 811
CurrentThread property, 746

D

Data members, class, 109
Data type(s), 24, 25
casting. *See* Casts
class as a, 111
conversion. *See* Type conversion
importance of, 35
object as a "universal", 309–310
promotion of, 59–60
reference. *See* Reference types
value (simple). *See* Value types
See also Type
Date and time, formatting, 695–700
custom placeholder characters for, table of, 699–700
format specifiers for, table of, 696
DateTime structure, 660, 695
Deadlock, 727, 745
Decimal .NET structure, 396, 397, 623, 630–634
constructors, 630
fields, table of, 635
methods, table of, 631–634
decimal value type, 36, 40–41, 396, 623, 630
literal, 46
DeclaringType property, 463
Decrement operator (– –), 29, 63, 64–65
overloading, 217
Decrement( ), 739–740
default #line directive option, 456
default statement, 89–91
and goto, 106–107
default(*type*) and type parameters, 522–523
DefaultCachePolicy property, 825
DefaultWebProxy property, 826
#define directive, 451, 452, 491
delegate keyword, 399, 409, 410
Delegates, 399–413
and anonymous functions, 408–409
and anonymous methods, 408–413
covariance and contravariance and, 406–408
declaring, 399–400
and events, 419, 420
generic, 527–530
and instance methods, 402–404
and lambda expressions, 413–419
and multicasting, 404–406
reasons to use, 408
System, list of, 617
Delegates, instantiating, 401
using method group conversion, 402, 403
Dequeue( ), 769, 770, 796, 797
descending contextual keyword, 550, 553
Destructors, 131–133
DialogResult enumeration, 856–857
Dictionaries, 749, 754, 787
Dictionary<TK,TV> class, 775, 779, 787–789
methods, table of some, 787
Dictionary<TK,TV>.KeyCollection class, 788
Dictionary<TK,TV>.ValueCollection class, 788
DictionaryBase class, 774
DictionaryEntry structure, 755, 763, 765, 810
DirectoryNotFoundException, 372, 373, 381
disable warning option, 457
Dispose( ), 374, 778
and the using statement, 608
Divide( ), 631
DivideByZeroException, 337, 342, 343, 347–348, 353
DivRem( ), 618
DllImport attribute, 609, 610
Documentation comments, 867–870
do-while loop, 85, 101–102, 105
Dot operator (.), 111, 119, 205, 332, 589
Double .NET structure, 396, 397, 623, 626
fields, table of, 630
methods, table of, 628–629
double value type, 25–26, 36, 38–39, 46, 396, 623
DoubleToInt64Bits( ), 655
DownloadData( ), 844
DownloadFile( ), 844
DownloadString( ), 844
DropDownItems property, 864
Dynamic method dispatch, 294

E

E field, 617
#elif directive, 451, 452, 454–455
else, 85–88
#else directive, 451, 454–455
Empty field, 430, 665
Encapsulation, 12, 20, 52, 165
Encoding class, 664
Encoding property, 843
#endif directive, 451, 452–453, 454
EndOfStream property, 381
EndOfStreamException, 390
#endregion directive, 451, 456
EndsWith( ), 552, 673, 675
Enqueue( ), 769, 770, 796, 797
Enter( ), 723
Entry property, 809–810
enum keyword, 332, 519
Enumerable class, 574, 577, 580, 684, 776
Enumerations, 36, 311, 332–335
  formatting, 700–701
  System, list of, 617
Enumerators, 749, 750, 808–811
Epsilon field, 628, 630
EqualityComparer property, 763
equals, 566, 568
Equals( ), 226, 306, 624, 626, 628, 631–632, 635, 640, 644, 659, 660, 754–755, 778
#error directive, 451, 456
Errors
  network, handling, 830–833
  runtime, 337, 501
  syntax, 22
  See also Exception handling
Escape sequences, character, 47
event statement, 419, 420, 424–425, 428
Event handlers, 419, 420–421
  anonymous methods as, 429, 430
  and event accessors, 424–429
  and form-based Windows programs, 853–856
  instance methods as, 422–423
  lambda expressions as, 429–430, 433–435
  .NET guidelines for 430–433
  static methods as, 422, 423–424
Event synchronization object, using an, 737–739
EventArgs class, 430, 432, 433, 854
EventHandler delegate, 432–433, 854
Events, 399, 419–435
  and generics, 544
  multicasting, 421–422
EventWaitHandle class, 737
Exception class, 337, 347, 351–355, 358, 359
Exception handling, 337–362
  blocks, general form of, 338, 349
  and creating derived exception classes, 348, 354–358
  inner, 352
  network, 830–833
  and uncaught exceptions, 341–343
  See also catch statement(s)
  See also try block(s)
Exceptions, standard built-in, 337, 352–354
  table of common, 353
ExceptionState property, 742, 743
ExceptWith( ), 798
exe file, 457, 480–481
Exists( ), 644, 652–653
Exit( ), 723
Exp( ), 618
explicit keyword, 235
Expression class, 581
Expression(s)
  nullable types in, 600–601
  spacing and parentheses in, using, 83
  trees, 581–582
  type conversion in, 59–62, 600–601
Extension methods, 7, 211, 564, 582–584
  and collections, 776
  corresponding to query keywords, 574–577
  definition of, 574
  general form of, 583
  the inheritance mechanism vs., 582–583
  query-related, 577–581
  and String objects, 684
extern
  used with methods, 544, 609–610
  used to provide an assembly alias, 451, 610–611

F

false, 42, 67
  overloading, 226–228, 231–235
False displayed as bool value output by WriteLine( ), 43, 640
FalseString field, 640
Field
  const, 606, 607
  definition of, 12, 109
  properties to manage access to, 254–259
  readonly, 606–607
  static, 208–209, 544
  unassigned or unused, 598
  volatile, 607
File Transfer Protocol (FTP), 823–824
File(s)
  I/O. *See* I/O, file
  position indicator (pointer), 390
FileAccess enumeration, 373
FileMode enumeration, 372
  values, table of, 372
FileNotFoundException, 372, 373, 381
FileStream class, 365, 372–378, 382, 384, 390
Finalize( ), 306, 659
finally block, 338, 349–351
Find( ), 644, 652–653, 784
FindAll( ), 644, 652
FindIndex( ), 644, 652
FindLast( ), 644, 784
FindLastIndex( ), 644
First property, 784
First( ) extension method, 578–579
fixed statement, using the, 588–589, 593
  and arrays, 591
  to create fixed-size buffers, 596–598
FixedSize( ), 757
Flags attribute, 700, 701
float value type, 25, 36, 38, 396, 623
  literal, 46
Floating-point(s), 25–26, 38–39
  literals, 46
  structures, 626–630
Floor( ), 618–619, 632
Flush( ), 364, 366, 376, 384
for loop, 28–29, 85, 92–99, 105
  and code blocks, 29, 30–31, 92
  variations, 94–99
foreach loop, 85, 102, 152–156, 749, 750, 754, 808, 809, 811
  to execute a query, 546, 548, 550
  using named iterator to control, 816
ForEach( ), 644, 653–654
Form, Windows, 849
  creating a, 849, 851
Form class, 849, 851
Format specifiers for WriteLine( ), 43–46

Format(), 685, 688–691
methods, table of, 689
FormatException, 397
Formatting, 684–701
and cultural settings, 685, 686, 695
date and time. *See* Date and time, formatting
enumerations, 700–701
and format providers, 685
and format specifiers, 685
using a picture format, 692
Formatting numeric data
custom, 45, 692
custom format placeholder characters for, table of, 695
format specifiers for, table of, 687
using String.Format(), 688–691
using ToString(), 691–692
using WriteLine(), 43–46, 686–688
Frank, Ed, 4
from clause, 547, 550
nested, 560–561
FromOACurrency(), 632
FTP (File Transfer Protocol) protocol, 823–824
FullName property, 462, 464
Func delegate types, predefined, 581, 582
Func<T,TResult> delegate, 574
Func<TInner, TKey> type, 575
Func<TOuter, TInner, TResult> type, 575
Func<TOuter, TKey> type, 575
Function
anonymous, 408–409, 413
members, 12, 109

G

Garbage collection, 131–133, 186, 596, 657–658, 848
and arrays, 137
and fixed, 588
GC class, 657–658
methods, table of, 658
Generic class
example with one type parameter, 496–499
example with two type parameters, 502–503
general form, 503
hierarchies, 537–540
instantiation process for, 543–544
overriding virtual methods in a, 540–542
partial, 604
reference to a, declaring a, 503
Generic method(s), 524–527, 544
and constraints, 527
and explicit type arguments, 527
general form, 526
Generics, 7, 310, 495–544
avoiding runtime errors through, 501
and casts, 496, 499, 501, 502
and code-bloat, 543–544
restrictions on using, 544
and type safety, 495, 496, 498, 499–502, 506, 526, 529, 544, 641, 649, 660, 750, 775, 800, 801
get accessor
and auto-implemented properties, 259–260, 262
and access modifiers, 261–264
for indexer, 246, 249, 320
for property, 254, 256, 261, 318–319
get contextual keyword, 33
Get(), 772
GetBits(), 632
GetByIndex(), 764, 765
GetBytes(), 655
GetConstructors(), 464, 472
GetCustomAttribute(), 485
GetCustomAttributes(), 464, 485
GetEnumerator(), 644, 752, 753, 754, 762, 763, 764, 765, 776, 777, 778, 787, 790, 792, 808, 809, 811
and iterators, 813–814, 816, 819
GetEvents(), 464
GetFields(), 464
GetFormat(), 662, 685
GetGeneration(), 658
GetGenericArguments(), 464
GetHashCode(), 226, 306, 624, 626, 628, 632, 636, 640, 644, 659, 754–755, 778
GetKey(), 764, 765
GetKeyList(), 764, 765
GetLength(), 644
GetLongLength(), 644
GetLowerBound(), 644
GetMembers(), 464
GetMethods(), 464, 465, 469
BindingFlags form of, 468–469, 483
GetNumericValue(), 636
GetParameters(), 465, 472
GetProperties(), 464
GetRange(), 757, 780
GetRequestStream(), 825
GetResponse(), 825, 826, 829, 830
exceptions, 830–831
GetResponseStream(), 827, 829, 830
exceptions, 831
GetTotalMemory(), 658
GetType(), 306, 501–502, 659
GetTypeCode(), 624, 626, 628, 632, 636, 640
GetTypes(), 476
GetUnicodeCategory(), 636
GetUpperBound(), 644
GetValue(), 645
GetValueList(), 764, 765
GetValues(), 834
global predefined identifier, 412–413
Gosling, James, 4, 5
goto, 85, 91, 105–107
group clause, 548, 550–551
using into with a, 561–563
GroupBy() extension method, 574, 576

H

Hash code, 762
Hash table, 762
HashSet<T> class, 779, 797–799
set operation methods, table of, 798
Hashtable class, 749, 755, 762–764, 774, 809
methods, table of common, 763
HasValue property, 599
hcp property, 763
Headers, accessing HTTP, 834–836
Headers property, 826, 827, 834, 835, 843
Heap, 596
Height property, 855, 863
Hejlsberg, Anders, 6
Hexadecimal literals, 47
hidden option for #line directive, 456
Hierarchical classification, 13, 269
High surrogate, 635
Host property, 833–834
HTTP (HyperText Transfer Protocol), 823–824, 826
cookies, accessing, 836–838
header, accessing, 834–836
HttpStatusCode, 823, 835
HttpWebRequest class, 822, 823, 824, 826–827, 837
HttpWebResponse class, 822, 823, 824, 826–827, 834–839
properties, table of, 834
HybridDictionary class, 774

ICloneable interface, 641, 660–662, 664, 755, 762, 764, 767, 769, 771
ICollection interface, 326, 641, 751–752, 753, 754, 755, 758, 762, 764, 767, 769, 771, 779, 783, 787, 789, 792, 794, 795, 836
ICollection<KeyValuePair<TK,TV>> interface, 777, 787, 789, 792
ICollection<T> interface, 775–776, 779, 783, 794, 796, 797
IComparable interface, 326, 534–536, 537, 624, 626, 630, 635, 640, 648–649, 659–660, 664
  implementing the, 801–803
IComparable<bool> interface, 640
IComparable<char> interface, 635
IComparable<decimal> interface, 630
IComparable<string> interface, 664
IComparable<T> interface, 536–537, 624, 626, 648–649, 659–660
  implementing the, 803–805
IComparer interface, 641, 751, 754, 789
  using an 805–806
IComparer<T> interface, 641, 775, 778
  using an, 806–808
IConvertible interface, 624, 626, 630, 635, 640, 660, 664
Identifiers, 33–34
IDeserializationCallback interface, 762, 783, 787, 797
IDictionary interface, 751, 753–754, 762, 764, 765, 787, 789, 792
  methods, table of, 753
IDictionary<TK,TV> interface, 775, 777, 787, 788, 789, 792
IDictionaryEnumerator interface, 751, 754, 762, 764, 809–811
IDisposable interface, 608, 614
IEEERemainder( ), 619
IEnumerable interface, 546, 550, 564, 641, 664, 750, 751, 752, 753, 754, 755, 762, 764, 767, 769, 771, 775, 776, 777, 778, 779, 783, 787, 789, 792, 794, 795, 797, 813, 816, 836
  implementing the, 811–812
IEnumerable<char> interface, 664
IEnumerable<IGrouping<TKey, TElement>>, 561
IEnumerable<KeyValuePair<TK,TV>> interface, 777, 787, 789, 792
IEnumerable<T> interface, 546, 547, 549–550, 566, 574, 576, 577, 750, 775, 776, 777, 778, 779, 783, 794, 795, 797, 818
IEnumerator interface, 326, 750, 751, 754, 778, 808, 809, 813
  implementing the, 811–812
IEnumerator<T> interface, 750, 775, 777, 778, 808, 818, 819
IEqualityComparer interface, 751, 754–755
IEqualityComparer<T> interface, 775, 778
IEquatable<bool> interface, 640
IEquatable<char> interface, 635
IEquatable<decimal> interface, 630
IEquatable<string> interface, 664
IEquatable<T> interface, 624, 626, 660
#if directive, 451, 452–454, 491
if statement, 27–28, 85–88
  and bool values, 43, 85
  and code blocks, 29–30, 85
if-else-if ladder, 87–88
IFormattable interface, 624, 626, 630, 662
IFormatProvider interface, 623, 662, 685
IGrouping<TKey, TElement>, 561
IHashCodeProvider interface, 751
IList interface, 641, 751, 752–753, 755, 779
  methods, table of, 752
IList<T> interface, 775, 776, 779
ImpersonationLevel property, 826
Implication operation, 69–70
implicit keyword, 235
Increment operator (++), 29, 63, 64–67
  overloading, 217–220
Increment( ), 739–740
Indentation style, 32
Indexers, 245–254
  abstract, 301
  and generics, 544
  interface, 320–322
  multidimensional, 252–254
  one-dimensional, 245–252
  overloading, 249–251
  overriding, 297
  and properties to create a custom array, using, 262–264
  read-only or write-only, creating, 249
  restrictions, 252
IndexOf( ), 157, 645, 673, 675–676, 752, 753, 757, 776, 780
IndexOf<T>( ), 645
IndexOfAny( ), 673
IndexOfKey( ), 764, 765, 792
IndexOfValue( ), 764, 765, 793
IndexOutOfRangeException, 141, 338–339, 342, 343, 353
Indirection, 586
  multiple, 594–595
Inheritance, 12, 13, 269–310
  basics of, 269–272
  and class member access, 272–276
  and constructors, 276–282, 288–289, 291–294
  and generic classes, 537–540
  and interfaces, 322–323
  multilevel hierarchy of, 272, 285–288
  name hiding and, 282–285, 323
  sealed to prevent, using, 305
  structures and, 327
  syntax for class, 271
Inherited named parameter, 491
Initialize( ), 645
InnerException property, 352
Insert( ), 682–683, 752, 753, 776
InsertRange( ), 756, 757, 780
Instance of a class, 109, 111
  See also Object(s)
Instance variables
  accessing, 111, 119
  and constructors, 127
  declaring, 111
  definition of, 12
  this to access hidden, using, 135
  as unique to their object, 111, 113–114
int value type, 24, 25–26, 36, 37, 38, 396, 623
Int16 .NET structure, 396, 397, 623
Int32 .NET structure, 396, 397, 623
Int64 .NET structure, 396, 397, 623
Int64BitsToDouble( ), 655
Integer(s), 23–24, 36–38
  literals, 46
  promotion, 59
Integer structures, 623–624
  methods, table of, 624–626
Interface(s), 311–326
  abstract class vs., 326
  and events, 429
  explicit implementation of, 323–326
  general form of, 312
  generic, 530–534
  implementing, 312–316
  indexers, 320–322
  and inheritance, 322–323
  .NET Framework standard, 326

Interface(s) *(continued)*
partial, 603
properties, 318–320
reference variables, 316–318
System, list of, 616
interface keyword, 311, 312
Interlocked class, 739
internal access modifier, 165, 166, 458
InternalsVisibleTo attribute, 605
Internet, 4–5, 821
access fundamentals, 823–830
and portability, 5
IntersectWith( ), 798
into
to create a continuation, using, 563–564
to create a group join, using, 569, 571–574
InvalidCastException, 353
InvalidOperationException, 599, 601, 767, 769, 794, 796, 830
Invoke( ), 469–471, 472
I/O, 363–398
array, 392–394
console, 364. 367–371
memory-based, 392–395
redirection, 364, 371, 381–383, 394
streams. *See* Streams
string, 394–395
I/O, file, 371–392
and binary data, 383–390
byte-oriented, 371–378
character-based, 372, 378–381
random access, 390–392
IOException, 364, 366, 367, 368, 372, 373, 374, 375, 376, 379, 381, 384, 385, 390, 831
IQueryable<T> interface, 574
is operator, 459–460, 461
IsAbstract property, 462, 464
IsAlive property, 710
IsArray property, 464
IsBackground property, 715
IsBusy property, 843
IsClass property, 462, 464
IsControl( ), 636
IsDefined( ), 464
IsDigit( ), 634, 636
IsEnum property, 464
ISerializable interface, 762, 783, 787, 797
IsFixedSize property, 641, 753, 754
IsFromCache property, 827, 835
IsGenericParameter property, 464
IsGenericType property, 464
IsHighSurrogate( ), 636
IsInfinity( ), 626, 628
IsLetter( ), 636
IsLetterOrDigit( ), 636
IsLittleEndian field, 654
IsLower( ), 636
IsLowSurrogate( ), 637
IsMutuallyAuthenticated property, 827, 835
IsNaN( ), 626, 628
IsNegativeInfinity( ), 627, 628
IsNumber( ), 637
IsPositiveInfinity( ), 627, 628
IsProperSubsetOf( ), 798
IsProperSupersetOf( ), 798
IsPunctuation( ), 637
IsReadOnly property, 641, 753, 754, 775, 794, 796
IsSeparator( ), 637
IsSubsetOf( ), 798
IsSupersetOf( ), 798
IsSurrogate( ), 637
IsSurrogatePair( ), 637
IsSymbol( ), 637
IsSynchronized property, 641, 751, 772
IsUpper( ), 637
IsWhiteSpace( ), 637
Items property, 864
Iterators, 749, 750, 813–819
generic, 818–819
named, 816–817
stopping, 815

J

Java
and C#, 3, 6
features lacking in, 5–6
history of, 4–5
Java Virtual Machine (JVM), 5
JIT compiler, 8, 15, 543
join clause, 566–569
using into with a, 569, 571–574
Join( ), 677, 678, 710–712, 741, 746, 747
extension method, 574, 577

K

KeepAlive( ), 658
Key, definition of, 753
Key property, 369–370, 561, 755, 778, 788, 791, 810
KeyChar property, 369–370
KeyComparer property, 763
KeysByTypeCollection<T> class, 779
Keys property, 754, 762. 764, 765, 777, 788, 790, 793
Keys<TK> property, 777
KeyValuePair<TK,TV> structure, 778, 788, 791, 792
Keywords, C#, 32, 33
Kill( ), 748

L

Label, 105
Lambda expressions, 7, 399, 408–409, 403–419
as event handlers, 429–430, 433–435
expression, 414–416
and expression trees, 581–582
and LINQ, 546
and query methods to create queries, 575–577
statement, 414, 416–419
Last property, 784
Last( ) extension method, 578–579
LastIndexOf( ), 157, 562, 645, 646, 673, 676–677, 757, 780
LastIndexOf<T>( ), 645, 646
LastIndexOfAny( ), 673
LastModified property, 835, 838
Left property, 855
Length property
of arrays, 148–151, 256–258, 642
of BitArray, 773
of Stream, 365
of String, 157, 665
let clause, 565–566
Library, .NET Framework class, 8, 20, 34, 38, 437, 613, 615
#line directive, 451, 457
LinkedList<T> class, 774, 779, 783–787
methods, table of a sampling of, 784–785
LinkedListNode<T> class, 783
LINQ (language-integrated query), 7, 545–584
and ADO.NET Datasets, 546
and anonymous types, 546, 569
and extension methods, 211, 546, 583
fundamentals, 546–551
and implicitly typed arrays, 151, 152
and implicitly typed variables, 52, 547, 571
and lambda expressions, 546
and object initializers, 199, 200, 261
and SQL, 545, 546, 581

and XML files, 545, 546
See also Query
List property, 783
List<T> class, 774, 779–783
methods, table of a sampling of, 780–781
ListDictionary class, 774
Literals, 35, 46–49
Little-endian format, 654
LoadFrom( ), 476
LocalPath property, 833–834
Location property, 852
lock, 605–606, 718–723, 728, 729, 730, 745
Log( ), 619
Log10( ), 619
Logical operators, 67–71
and nullable objects, 602–603
overloading the, 229–235
long value type, 36, 37, 396, 623
literal, 46
LongLength property, 642
Loops
break to exit, using, 102–104
do-while, 85, 101–102, 105
for. *See* for loop
foreach. *See* foreach loop
infinite, 98
while, 85, 99–101, 105
Low surrogate, 635

M

M and m suffixes to specify a decimal, 40, 46
Main( ), 21, 110, 116, 200–202, 205, 480–481
and command-line arguments, 200–202
MainMenu class, 859, 860, 862, 863, 864
MainMenuStrip property, 864
Managed code, 9, 585, 586
Manifest, assembly, 457
ManualResetEvent class, 737–739
Math class, 38, 39, 193, 617–623
methods, table of, 618–621
Max( ), 619
extension method, 578–579
MaxGeneration property, 657
MaxValue field, 624, 628, 630, 635
Member(s), class, 12, 109
access and inheritance, 272–276
controlling access to, 110, 165–172
dot operator to access, 111, 119, 205
static, 205
MemberInfo class, 463–464, 465, 472, 485
MemberType property, 463
MemberTypes enumeration, 463
MemberwiseClone( ), 306, 659, 660, 662
Memory allocation
using new, 115, 130, 131
using stackalloc, 596
Memory management, 657–658
MemoryStream class, 365, 392–394
Menu class, 859, 860
Menu property, 859, 861
Menu(s), Windows, 849, 852, 859–866
creating a new-style, 863–866
creating traditional, 859–863
MenuItem class, 859, 860, 864
MenuItems property, 859, 860, 864
MenuStrip class, 859, 863, 864
Message box, 856–858
Message property of Exception, 351–352
MessageBox class, 856
MessageBoxButtons enumeration, 856
Metadata, 8, 483
type, 457
MetadataToken property, 463
Method property, 826, 835
Method(s), 12, 21, 116–126
abstract, 301–304, 311
anonymous, 408–413, 417, 429, 430
base to access hidden, using, 283, 284–285
calling, 119, 469–471
class factory, 186–187
conditional, 491–492
and covariance and contravariance, 406–408
delegates and, 399–413
dispatch, dynamic, 294
dot operator (.) and, 111, 119, 205
extension. *See* Extension methods
extern, 609–610
general form of, 116
generic. *See* Generic method(s)
group conversion, 402, 403
and interfaces, 311–312, 313
and multicasting, 404–406
operator, 213–214, 220–224
overloading, 188–193, 542–543
overriding. *See* Overriding, method
and parameters. *See* Parameter(s)
partial, 7, 604–605
passing objects (reference types) to, 172–176
private members accessed by public, 165, 167, 168, 171
recursive, 202–205, 354
reflection to obtain information about, using, 465–469
return type of, 121
returning arrays from, 187–188
returning from, 119–120
returning objects from, 185–188
returning a value from, 120–122
returning more than one value from, 176, 178–181
scope defined by, 52–54
signature, 193
static, 21, 205, 206–208, 301
unreachable code and, 126
virtual, 291, 294–301, 305, 540–542
MethodBase class, 465, 472
MethodImplAttribute attribute, 728–729
MethodImplOptions.Synchronized attribute, 728
MethodInfo class, 465, 469, 472
MFC (Microsoft Foundation Classes), 848
Microsoft Intermediate Language (MSIL), 8, 15, 457, 543
Min( ), 619–620
extension method, 578–579
MinusOne field, 635
MinValue field, 624, 628, 630, 635
Modifiers property, 369–370
Module property, 463
Modulus operator (%), 63–64
Monitor class, 723
MoveNext( ), 754, 778, 808, 809, 810, 811, 812
MSIL (Microsoft Intermediate Language), 8, 15, 457, 543
Multicasting
delegates and, 404–406
and events, 421–422
Multiply( ), 632
Multitasking, 703–704
process-based, using, 747–748
thread-based vs. process-based, 703

Multithreaded programming, 605–606 703–748
and deadlock, 727, 745
and race conditions, 727
synchronization and. *See* Synchronization
and threads. *See* Thread(s)
tips for effective, 747
Mutex, 730–733, 745
named, 733
MutexSecurity, 733

N

Name hiding
and code blocks, 54
and inheritance, 282–285, 323
Name property, 463, 465, 707, 836–837
namespace keyword, 438
Namespace property, 464
Namespace(s), 20, 22–23, 437–451
additive characteristic of, 444–446
alias qualifier (::), 447–451
declaring, 438
default global, 437, 447, 450
nested, 446–447
qualification, 22–23, 438, 446, 447
NameValueCollection class, 774
NaN field, 628, 630
Narrowing conversion, 56–58
Naughton, Patrick, 4
Negate( ), 632
Negative numbers, representation of, 37, 81
NegativeInfinity field, 628, 630
.NET Framework, 3, 6, 7–8
assemblies, 457–458
class library, 8, 20, 34, 38, 437, 613, 615
event guidelines, 430–433
I/O system, 363, 364, 365, 371
standard interfaces, 326
structures for the built-in numeric types, 396–397, 623–641
Windows Forms subsystem, 847, 848, 850
.NET Reflection API, 463, 465, 476
Networking, Internet, 821–845
and handling errors, 830–833
simple example program for, 828–830
web crawler program for, 839–842
new, 114–115, 128, 130–131, 137, 138, 140, 141, 327, 328, 348
to hide base class member, 282–283, 285
to hide interface member, 323
and implicitly typed arrays, 151, 152
new{ } to create an anonymous type, 569
new( ), 504, 516–517, 521
Next property, 783, 787
Next( ), 657
NextBytes( ), 657
NextDouble( ), 657
NOT operator
bitwise unary (~), 73, 78–79
logical unary (!), 67, 68, 69–70, 229–231
Not( ), 772
NotSupportedException, 364, 372, 374, 375, 390, 752, 776, 825, 830
Null
coalescing operator (??), 601–602
reference, 353
null value, 353, 518, 522, 598, 599, 601, 602, 603
Nullable type, 36, 518, 519, 598–603
Nullable<T> class, 598–599
NullReferenceException, 353–354
NumberStyles enumeration, 623
Nybble, 240

O

Oak, 4
Object(s), 12, 109
attributes of, obtaining the, 485–486
class factory to construct, using a, 186–187
copies of, deep vs. shallow, 660, 662
creating, 111, 114–115
to methods, passing, 172–176
reflection for runtime instantiation of, using, 471–475
returning, 185–188
Object class, 659, 660
object class, 305–310, 467, 495–496, 749–750
methods defined by, table of, 306, 659
reference to value type, 307–309
Object initialization
with another object, 195–198
with constructor, 126–130
with object initialization syntax, 199–200, 260–261
Object.Equals( ), 226
Object.HashCode( ), 226
Object-oriented programming (OOP), 4, 11–13, 165
ObjectDisposedException, 374, 375, 376, 831
Obsolete built-in attribute, 491, 493
OfType( ) extension method, 752
on, 566, 568
One field, 635
One's complement (unary NOT) operator (~), 73, 78–79
OpenRead( ), 844
OpenWrite( ), 844
operator keyword, 213
Operator overloading, 213–244, 543
binary, 213–216, 220–224
logical, 229–235
and overloading an operator method, 220–224
relational, 224–226
restrictions, 239–240
and true and false, 226–229, 231–235
unary, 213–214, 216–220
Operator(s)
? conditional, 82–83
arithmetic, 24, 63–67
arrow (–>), 589
assignment. *See* Assignment operator(s)
bitwise, 73–81
conversion. *See* Conversion operators
decrement. *See* Decrement operator (– –)
and generics, 544
implication, 69–70
increment. *See* Increment operator (++)
lambda (=>), 413–414
logical. *See* Logical operators
methods, 213–214, 220–224
parentheses and, 83
pointer, 587
precedence, table of, 84
relational. *See* Relational operators
OR operator (|)
bitwise, 73, 76–77
logical, 67, 68, 69–70, 71, 229–235
OR operator, short-circuit or conditional (||), 67, 70, 71, 229, 231–235
Or( ), 772
orderby clause, 552–556

OrderBy() extension method, 574
OrderedDictionary class, 774
out parameter modifier, 176, 178–181, 192–193, 214, 252, 261
OutOfMemoryException, 353
Overflow in arithmetic computations, 360
OverflowException, 353, 397
  and checked and unchecked, 360–362
Overloading
  constructors, 194–199, 543
  indexers, 249–251, 543
  methods, 188–193, 542–543
  operators. *See* Operator overloading
override keyword, 294, 297
Overriding, method, 294–304
  and dynamic method dispatch, 294
  and virtual methods in a generic class, 540–541

P

PadLeft(), 681–682
PadRight(), 681–682
Parameter(s), 116
  and constructors, 128
  and methods, 122–126
  modifiers, ref and out, 176–182
  and overloaded methods, 188–193
  positional vs. named, 487–490
  type. *See* Type parameter(s)
  variable-length, 182–185
ParameterInfo class, 465
ParameterizedThreadStart delegate, 713
ParameterType property, 465
params parameter modifier, 182–185, 193
Parse() methods, 396–398, 624, 625, 627, 628–629, 632, 638, 640
partial contextual keyword, 33, 603, 604
PathAndQuery property, 833–834
PathTooLongException, 372, 373
Peek(), 366, 767, 769, 770, 794, 795, 796, 797
Permits, semaphore, 734
PI field, 617
Point structure, 852
Pointer(s), 544, 585–595
  arithmetic, 589–591, 592
  comparisons, 591
  indexing, 592–593
  multiple indirection and, 594–595
  operators, 587, 589
  referent type of, 586, 590, 591
  and strings, 593–594
Polymorphism, 12–13
  and overridden methods, runtime, 294, 297
  and interfaces, 311
  and overloaded methods, 188, 193
Pop(), 767, 794, 795
Port property, 833–834
Portability, 5, 8, 36
Position property, 365, 392
PositiveInfinity field, 628, 630
Pow(), 620, 621–623
#pragma directive, 451, 457
PreAuthenticate property, 826
Predicate delegate, 652–653
Preprocessor directives, 451–457
Previous property, 783, 787
Primitive types, 35
Priority property, 715
private access specifier, 165–167
  and inheritance, 276
Private class members and inheritance, 272–275
Projection initializer, 571
Process class, 747
Process, definition of, 703
Programming
  mixed-language, 5, 7, 8, 596
  multithreaded. *See* Multithreaded programming
  object-oriented (OOP), 4, 11–13
  structured, 3, 11, 12
Properties, 245, 254–268, 276
  access private base class members using, 273–275
  abstract, 301
  auto-implemented, 7, 255, 259–260, 319
  and generics, 544
  and indexers to create a custom array, using, 264–268
  interface, 318–320
  and object initializers, 260–261
  overridden, 297
  read-only or write-only, creating, 256
  restrictions on, 261
  virtual, 297
protected access specifier, 165, 273, 275–276, 458
Protocol, network, 823
  pluggable, 823–824, 829
ProtocolVersion property, 835
ProtocolViolationException, 830, 831
Proxy property, 826, 843
public access specifier, 110, 116, 165–167
  and inheritance, 276
Pulse(), 723–727
PulseAll(), 723–724
Push(), 767, 794, 795

Q

Queue class, 749, 755, 769–771
  methods, table of, 770
Queue<T> class, 779, 795–797
  methods, table of, 797
Query
  contextual keywords, list of, 550
  continuation, 563–564
  data source, 547, 549–550
  data types, relationship among, 549–550
  definition of a, 546
  determining what is obtained by a, 548, 550, 556–559
  general form of a, 550, 551
  and group joins, 571–574
  group results of a, 561–563
  joining two data sources in a, 566–569
  property, 833–834
  range variable, 547, 549–550, 556
  sort results of a, 552–556
  values returned by a, filter, 548, 551–552
  variable, 547, 548, 549, 550
  variable in a, create a, 565–566
Query, creating a
  using the query methods, 574–577
  using the query syntax, 547–548, 577
Query execution, 546, 548–549, 550
  deferred vs. immediate, 580–581
Query methods, 574–579
  corresponding to the query keywords, 574–575
  and lambda expressions to create queries, using, 575–577
Queryable class, 574, 684
QueryString property, 843
Quicksort algorithm, 205

R

Race condition, 727
Random access file I/O, 390–392

Random class, 656–657
methods, table of, 657
Random numbers, generating, 656–657
Rank property, 642
Read( ), 364, 366, 367–368, 371, 374, 385
ReadBlock( ), 366
ReadByte( ), 364, 374–375, 385, 830
ReadKey( ), 367, 368, 369–371
ReadLine( ), 366, 367, 368–369, 371, 381, 394, 395
readonly, 606–607
ReadOnly( ), 757
ReadOnlyCollectionBase class, 774
ReadString( ), 384, 385
ReadTimeout property, 365
ReadToEnd( ), 366
Recursion, 202–205, 354
ref parameter modifier, 176–178, 181–182, 192–193, 214, 252, 261
Reference types, 35, 115, 130
default value of, 127, 522
to methods, passing, 172–176
using ref and out with, 181–182
Reference variable(s)
derived class objects and base class, 289–294, 301
and assignment, 115–116, 146–147
declaring, 114–115
interface, 316–318
ReferenceEquals( ), 306, 659
ReflectedType property, 463
Reflection, 459, 463–483
API, .NET, 463, 465, 476
and assemblies, 475–481
to call methods, using, 469–471
and fully automated type discovery, 481–483
to obtain a type's constructors, 471–475
#region directive, 451, 456
RegisterPrefix( ), 824
Relational operators, 27, 43, 67–68, 226–228
and nullable objects, 602
Release( ), 734
ReleaseMutex( ), 730, 733
Remainder( ), 632
remove event accessor, 425, 428
Remove( ), 682–683, 752, 753, 776, 777, 784–785, 787, 790, 793, 794, 796
RemoveAt( ), 752, 776, 793
RemoveFirst( ), 785
RemoveLast( ), 785
RemoveMemoryPressure( ), 658
RemoveRange( ), 756, 757, 780
Replace( ), 401, 682–683
RequestUri property, 826
ReRegisterForFinalize( ), 658
Reset( ), 737, 739, 754, 778, 808, 811
ResetAbort( ), 743–745
Resize<T>( ), 646
Response property, 831
ResponseHeaders property, 843
ResponseUri property, 827, 835
restore warning option, 457
return statement, 85, 105, 119–120, 176, 410, 411, 417, 814
ReturnType property, 465
Reverse( ), 646, 650, 756, 757, 780
Right property, 855
Ritchie, Dennis, 3
Round( ), 620, 632–633
Run( ), 851
Runtime type identification (RTTI), 459–463

S

Sample( ), 657
SByte .NET structure, 396, 397, 623
sbyte value type, 36, 37, 38, 396, 623
Scheme property, 833–834
Scopes, 52–54, 438, 457
sealed keyword, 305
Security, 8, 9
SecurityException, 372
Seek( ), 364, 365, 390–392, 393–394
SeekOrigin enumeration, 390
select clause, 548, 550, 556–559, 561, 569
using into with a, 563
Select( ) extension method, 574, 575–576
Selection statements, 85–92
Semaphore, 730, 734–737
named, 736–737
SemaphoreSecurity, 737
SerializationInfo, 352
Server property, 835
set accessor
and accessor modifiers, 261–264
and auto-implemented properties, 259–260, 262–264
for indexer, 246, 249, 320
for property, 254–255, 256, 318–319
set contextual keyword, 33
Set( ), 737, 739, 772
SetAll( ), 772
SetByIndex( ), 764, 765
SetEquals( ), 798
SetError( ), 382
SetIn( ), 382
SetOut( ), 382–383
SetRange( ), 756, 757
SetValue( ), 646
Sheridan, Mike, 4
Shift operators, bitwise, 73, 79–81
short value type, 36, 37, 38, 396, 623
Show( ), 856–857
Sign flag, 37
Sign( ), 620
Signature of a method, 193
Simple types, 35
Sin( ), 40, 620
Single .NET structure, 396, 397, 623, 626
fields, table of, 628
methods, table of, 626–627
Sinh( ), 620
sizeof operator, 595
Sleep( ), 706, 746
Sort( ), 646, 647, 648, 756, 757, 760–761, 780–781
Sort<T>( ), 646, 647
Sort<TK,TV>( ), 647
SortedDictionary<TK,TV> class, 779, 789–791
methods, table of some, 790
performance characteristics, 792
SortedDictionary<TK,TV> .KeyCollection class, 790
SortedDictionary<TK,TV> .ValueCollection class, 790
SortedList class, 755, 764–767
methods, table of common, 765
SortedList<TK,TV> class, 779, 792–794
methods, table of some, 792–793
performance characteristics of, 792
Split( ), 677–680
SQL (Structured Query Language), 545, 546, 581
Sqrt( ), 38–39, 206, 621
Stack
definition of, 13, 168, 767
memory region, 596
Stack class, 749, 755, 767–769
methods, table of, 767
Stack<T> class, 779, 794–795
methods, table of, 795
stackalloc, 596
StackOverflowException, 353, 354, 596

StackTrace property of Exception, 351–352
Start(), 705, 747–748
parameterized version of, 713
StartsWith(), 673, 677
Statements, 21, 31
empty, 98
Statements, control, 26
iteration, 85, 92–102
jump, 85, 102–107
selection, 85–92
STAThread attribute, 850, 851
static, 21, 205–212, 252, 294
Status property, 831
StatusCode property, 835
StatusDescription property, 835
Stream(s)
binary, 367
byte, 363, 372–378
character, 363, 364, 365, 367, 378–381
classes, byte, 365
definition of, 363
memory-based, 392–394
predefined, 363–364
redirecting standard, 364, 371, 381–383, 394
wrapper classes, character, 365–367
Stream class, 364–365, 372, 393
byte steam classes derived from, 365
methods, table of common, 364
properties, table of, 365
StreamingContext, 352
StreamReader class, 367, 378, 380–381, 608
StreamWriter class, 367, 378–380, 382
string
alias for System.String, 159, 663
class. *See* String class
data type, 156, 617, 663, 664
String class, 156–163, 663–684
comparison methods, table of common, 666–667
constructors, 664
indexer, 665
search methods, table of, 675–677
string handling methods, table of common, 157
String(s)
arrays of, 160–161
comparing, 158, 665–670
concatenating, 159, 670–672
constructing, 156–157, 664–665
extension methods, 684
formatting. *See* Formatting
immutability of, 161–162, 663
indexing, 157
inserting, 682–683
I/O using, 394–395
joining, 677, 678
Length property of, 157, 665
literals, 48–49, 156, 665
numeric, converting, 396–398
as objects, 156, 663
padding, 681–682
pointers and, 593–594
relational operators and, 665
removing, 682–683
replacing, 682–683
searching, 672–677
splitting, 677–680
switch statements and, 162–163
tokenizing, 679–680
trimming, 681–682
String.Format(). *See* Format()
StringBuilder class, 162, 395, 663
StringCollection class, 774
StringComparison enumeration, 669–670
StringDictionary class, 774
StringReader class, 367, 394–395
StringSplitOptions enumeration, 679
StringWriter class, 367, 394, 395
Stroustrup, Bjarne, 4, 5
struct keyword, 326, 504, 517, 519, 521
Structures, 36, 311, 326–331
C# vs. C++, 331
corresponding to the built-in value types, .NET, 396–397, 623–641
default value of, 522
efficiency of, 326, 330–331
and fixed-size buffers, 596–598
generic, 523–524
partial, 603
using pointers with, 589
System, list of, 616
Subclass, 275
Substring( ), 161–162, 562, 684
Subtract(), 633
Sum( ) extension method, 578–579
Sun Microsystems, 4
Superclass, 275
SuppressFinalize(), 658
Suspend(), 745
switch statement, 85, 88–92
enumerations and, 89, 332, 335
goto and, 106–107
and no fall-through rule, 91–92, 107
and strings, 89, 162–163
Symbol, #define directive, 452
SymmetricExceptWith(), 798
Synchronization, 704, 717–723
deadlock, 727, 745
interprocess, 733, 737
using an event synchronization object, 737–739
using the Interlocked class, 739–740
using a lock statement, 718–722, 723, 729
using the MethodImplAttribute attribute, 728–729
using Monitor methods, 723–727
using a mutex, 730–733
and race conditions, 727
using a semaphore, 730, 734–737
SynchronizationLockException, 724
Synchronized(), 756, 757, 763, 764, 765, 767, 770, 772
SynchronizedCollection<T> class, 779
SynchronizedKeyCollection<K,V> class, 779
SynchronizedReadOnlyCollection<T> class, 779
Synchronous vs. asynchronous Internet communication, 824
SyncRoot property, 642, 751, 772
Syntax errors, 22
System namespace, 20, 337, 352, 353, 363, 364, 396, 437, 581, 613, 615, 664, 833, 850
members, lists of, 615–617
System.Attribute, 483
System.AttributeUsageAttribute class, 491
System.Collections namespace, 641, 750, 755, 774
See also Collection(s)
System.Collections.Generic, 546, 641, 750, 774, 778, 779
System.Collections.ObjectModel, 750
System.Collections.Specialized namespace, 750, 774
System.ComponentModel.Component class, 849
System.Delegate, 408
System.Diagnostics namespace, 491, 747
System.Diagnostics.ConditionalAttribute class, 491
System.Drawing namespace, 852

System.Exception class. *See* Exception class
System.Globalization namespace, 623, 638, 666, 667
System.IDisposable interface, 608
System.IO namespace, 364, 365, 372, 437
System.IO.Stream class. *See* Stream class
System.Linq namespace, 547, 584, 684
System.Linq.Enumerable, 574, 752
System.Linq.Queryable, 574, 664, 752
System.Linq.Expressions.Expression<T>, 581
System.Math class, 38, 193, 206
  See also Math class
System.Net namespace, 821
  classes, list of, 822
  enumerations, list of, 823
  interfaces, list of, 822
  Internet networking using members of the, 821–845
System.Net.Mail namespace, 821
System.Net.Security namespace, 821
System.Net.Sockets namespace, 821
System.Nullable<T> class, 598–599
System.Object class, 306
  See also object class
System.ObsoleteAttribute, 493
System.Predicate, 652
System.Reflection namespace, 463, 465
System.Reflection.MemberInfo, 463
  See also MemberInfo class
System.Runtime.CompilerServices namespace, 728
System.Runtime.InteropServices namespace, 609
System.Runtime.Serialization namespace, 352
System.Security.SecurityException, 830
System.ServiceModel.UriSchemeKeyedCollection, 779
System.String class, 159, 617, 663
  See also String class
System.Text namespace, 162, 663, 664
System.Threading namespace, 704, 723
System.Threading.Mutex class, 730
System.Threading.Semaphore class, 734
System.Type, 462, 463
  See also Type class
System.Web namespace, 821
System.Windows.Forms, 847, 850, 851
System.Windows.Forms.Control class. *See* Control class
SystemException, 337, 352

T

Tan( ), 40, 621
Tanh( ), 621
TargetSite property of Exception, 351–352
Text property. 850–851, 852
TextReader class, 365–367, 369, 378, 382, 394, 608
  methods defined by, table of, 366
TextWriter class, 365, 366–367, 371, 378, 382, 394
this, 133–135, 206
  and overloaded constructors, 197–199
Thread class, 704, 705, 706, 710, 715, 745, 746
  constructors, 704–705, 713
Thread.Resume( ), 745
Thread.Suspend( ), 745
Thread.WaitHandle class, 730, 734
Thread(s)
  argument to, passing an, 713–714
  communication, 723–727
  creating and starting, 704–710
  deadlock, 727, 745
  definition of, 703
  end of, determining the, 710–712
  foreground and background, 705, 715
  main, 704, 706–707, 746–747
  multiple child, 606, 708–710
  possible states of, 704
  priorities, 715–717
  and race conditions, 727
  state, determining, 745–746
  suspending and resuming, 745
  synchronization. *See* Synchronization
  terminating, 741–745
ThreadAbortException, 741, 742, 743
Threading models (Windows), 851
ThreadPriority enumeration, 715
ThreadStart delegate, 704–705, 713
ThreadState
  enumeration, 746
  property, 745
ThreadStateException, 705, 711
throw, 85, 338, 347–349
Timeout property, 826
ToArray( ), 757–758, 761–762, 767, 770, 781, 795, 797
  extension method, 580–581
ToBoolean( ), 655
ToByte( ), 633
ToChar( ), 655
ToDouble( ), 633, 655
ToInt16( ), 633, 655
ToInt32( ), 633, 655
ToInt64( ), 633, 655
Token, 679
ToList( ) extension method, 580–581
ToLower( ), 157, 638, 683
ToLowerInvariant( ), 638, 683
ToOACurrency( ), 633
ToolStrip class, 859, 863
ToolStripMenuItem class, 859, 864
Top property, 855
ToSByte( ), 633
ToSingle( ), 633, 655
ToString( ), 306–307, 309, 351, 395, 625, 626, 627, 629, 633–634, 638, 640, 656, 659, 662, 671, 672
  used to format data, 685, 688, 691–692, 697
ToUInt16( ), 634, 656
ToUInt32( ), 634, 656
ToUInt64( ), 634, 656
ToUpper( ), 157, 634, 688, 683
ToUpperInvariant( ), 638, 683
Trim( ), 681–682
TrimExcess( ), 781, 793, 795, 797
TrimToSize( ), 758, 765, 770
true, 42, 67
  overloading, 226–229, 231–235
True and false in C#, 42
True displayed as bool value output by WriteLine( ), 43
TrueForAll<T>( ), 647
TrueString field, 640
Truncate( ), 621, 634
try block(s), 338–344
  nested, 346–347
TryEnter( ), 723
TryGetValue( ), 777
TryParse( ), 398, 625, 626, 627, 629, 634, 638, 641
Two's complement, 37
Type(s)
  anonymous, 7, 199, 200, 261, 569–571
  casting. *See* Casts
  checking, 35, 49, 309
  closed, 499

constrained, 504. *See also* Constraint(s)
constructed. *See* Constructed type
data. *See* Data types
extensibility, 213, 240
identification, runtime, 459–463
inference, 416, 526, 527, 549–550
metadata, 457
nullable, 36, 518, 519
numeric, .NET structures for the built-in, 397–397
open, 499
parameterized, 495
promotion, 59–60
reference. *See* Reference types
safety and generics, 495, 496, 498, 499–502, 506, 526, 529, 544, 641, 660, 750, 775, 800, 801
value. *See* Value types
Type argument(s), 498, 499
and constraints, 503–521. *See also* Constraint(s)
and pointer types, 544
to call a generic method, using explicit, 527
Type class, 462, 463, 465, 485
commonly used methods defined by the, list of, 464
commonly used properties defined by the, list of, 464
Type conversion, 42, 55–62
and casting, 55, 56–58, 60–62
implicit (automatic), 55–56, 59, 190–192
Type parameter(s), 497–498, 544
comparing instances of a, 534–537
and constraints, 503–504
creating a default value of a, 522–523
and generic class hierarchies, 537–540
and generic delegates, 527
and generic interfaces, 530, 533, 534
and generic methods, 524–526
overloading methods that use, 542–543
relationship between two, establishing a, 520
TypeCode enumeration, 624, 626, 628, 632, 636, 640
typeof operator, 462–463, 467, 476

U

uint value type, 36, 37, 38, 396, 623
literal, 46
UInt16 .NET structure, 396, 397, 623
UInt32 .NET structure, 396, 397, 623
UInt64 .NET structure, 396, 397, 623
ulong value type, 36, 37, 396, 623
literal, 46
Unboxing, 307–309
unchecked keyword, 360–362
#undef directive, 451, 455
Unicode, 41, 363, 378
UTF–16 and UTF–32, 634–635
UnicodeCategory enumeration, 636
Uniform Resource Identifier (URI), 823
constructing a, 833
dissecting a, 833–834
Uniform Resource Locator (URL), 823
UnionWith(), 798
Unmanaged code, 9, 585, 586
UnmanagedMemoryStream class, 365
/unsafe compiler option, 586
unsafe keyword, using, 587–588
UploadData(), 844, 845
UploadFile(), 845
UploadString(), 845
UploadValues(), 845
Uri class, 823, 833–834
UriFormatException, 825, 830
UseDefaultCredentials property, 826, 843
ushort value type, 36, 37, 38, 396, 623
using directive
to bring a namespace into view, 20, 22–23, 441–443
to create an alias, 443–444
using statement, 374, 607–608

V

ValidOn property, 491
value
contextual keyword, 33
implicit parameter, 246, 248–249, 255, 425. 428
Value property, 599, 755, 778, 783, 788, 791, 809, 836–837
Value type(s), 35–36, 115
boxing and unboxing, 307–309
and call-by-reference, 176–181
constraints, 504, 517, 519
default value of, 127, 522
new and, 130–131
null and, 518
passed to methods, 174–175
structures as, 311, 326, 330, 331
structures corresponding to the built-in numeric, .NET, 396–397, 623–641
table of, 36
Values property, 754, 762, 765, 777, 788, 790, 793
Values<TV> property, 777
var keyword, 51, 151, 153, 547, 548, 550, 571
Variable(s)
captured, 412–413, 416
const, 607
declaration, 23–24, 25, 28, 49, 53
definition of, 23
dynamic initialization of, 50–51
implicitly typed, 7, 25, 51–52, 547, 569, 571
initializing, 50–51, 53–54
instance. *See* Instance variables
local, 49, 52–53
member, 12
names, rules governing, 33–34
outer, 412–413, 416
pointer. *See* Pointer(s)
query, 547, 548
reference. *See* Reference variable(s)
scope and lifetime of, 52–54
static, 205–206, 208–209
volatile, 607
Version class, 835
virtual keyword, 294, 297, 301
Virtual
methods, 291, 294–301, 305, 540–542
properties, 297
Visual C++ 2008 Express Edition, 15
Visual C# 2008 Express Edition, 14, 15
Visual Studio IDE (Integrated Development Environment), 14, 15–19, 848, 851, 852, 867
void, 21, 116
methods, returning from, 119–120
volatile modifier, 607
vsvars32.bat file, 15

W

Wait(), 723–727, 746
WaitForExit(), 748
WaitForPendingFinalizers(), 658
WaitHandle class, 730, 734
WaitOne(), 730, 733, 734, 737, 739

#warning directive, 451, 456
warning #pragma option, 457
Warth, Chris, 4
Web crawler
  definition of, 839
  example program, 839–842
WebClient class, 822, 823, 842–845
  methods, table of synchronous, 844–845
  properties, table of, 843
WebException, 830–831, 842
WebExceptionStatus enumeration, 831
WebHeaderCollection collection, 834
WebRequest class, 823, 824–826, 829, 830, 842, 845
  methods, table of synchronous communication, 825
  properties, table of, 825–826
WebResponse class, 823, 824, 826, 830, 831, 834, 842, 845
  methods, table of commonly used, 827
  properties, table of, 827
where clause, 504, 505, 512, 517, 520, 521
  in a LINQ query, 547–548, 551–552
Where( ) extension method, 574, 575–576
while loop, 85, 99–101, 105
Widening conversion, 55, 56
Width property, 855, 863
Window, form-based
  changing default size of, 863
  illustration of skeletal, 850
Windows, 6, 8, 9
  API, 847–848, 851
  controls, 849, 852, 855
  form. *See* Form, Windows
  program, compiling, 851
Windows application, form-based
  adding button to, 852–853
  adding menu to, 859–866
  compiling, 851
  message handling, 853–856
  skeleton, 849–851
  threading model, 851
  using message box in, 856–858
Windows Forms .NET subsystem, 847, 848, 850
Windows programming
  form-based, 847–866
  history of, 847–848
  and message-based interaction, 849
  using Visual Studio, 848
Write( ), 24–25, 364, 366, 371, 375, 384
WriteByte( ), 364, 375
WriteLine( ), 21, 24–25, 26, 41, 43, 48, 307, 351, 366, 371, 394, 395, 396, 640
  formatted output version of, 43–46, 685
  using a method within, 122
WriteTimeout property, 365

X

X property, 852
XML, 545, 546
XML comment, 867
  tags, table of, 868
  XML file containing, 869–870
XOR (exclusive OR) operator (^)
  bitwise, 73, 77–78
  logical, 67, 68
Xor( ), 772

Y

Y property, 852
yield
  break statement, 815
  contextual keyword, 33, 814
  return statement, 814
  statements, multiple, 815–816

Z

Zero field, 635