Adrian Kingsley-Hughes Kathie Kingsley-Hughes

Adrian Kingsley-Hughes Kathie Kingsley-Hughes

**C# 2005 Programmer’s Reference**

Published by

**Wiley Publishing, Inc.**

10475 Crosspoint Boulevard

Indianapolis, IN 46256

[www.wiley.com](http://www.wiley.com/)

Copyright © 2007 by Wiley Publishing, Inc., Indianapolis, Indiana Published simultaneously in Canada

ISBN-13: 978-0-470-04641-8

ISBN-10: 0-470-04641-4

Manufactured in the United States of America 10 9 8 7 6 5 4 3 2 1

1B/RZ/RQ/QW/IN

Library of Congress Cataloging-in-Publication Data

Kingsley-Hughes, Adrian.

C# 2005 programmer’s reference / Adrian Kingsley-Hughes, Kathie Kingsley-Hughes.

p. cm.

ISBN-13: 978-0-470-04641-8 (paper/website)

ISBN-10: 0-470-04641-4 (paper/website)

1. C# (Computer program language) I. Kingsley-Hughes, Kathie. II. Title. QA76.73.C154K575 2006

005.13’3--dc22

2006030327

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Legal Department, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, (317) 572-3447, fax (317) 572-4355, or online at [http://www.wiley.com/go/permissions.](http://www.wiley.com/go/permissions)

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND THE AUTHOR MAKE NO REPRE- SENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION WAR- RANTIES OF FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES OR PROMOTIONAL MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY SITUATION. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER IS NOT ENGAGED IN RENDERING LEGAL, ACCOUNTING, OR OTHER PROFESSIONAL SERVICES. IF PROFES- SIONAL ASSISTANCE IS REQUIRED, THE SERVICES OF A COMPETENT PROFESSIONAL PERSON SHOULD BE SOUGHT. NEITHER THE PUBLISHER NOR THE AUTHOR SHALL BE LIABLE FOR DAMAGES ARISING HERE- FROM. THE FACT THAT AN ORGANIZATION OR WEBSITE IS REFERRED TO IN THIS WORK AS A CITATION AND/OR A POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE AUTHOR OR THE PUBLISHER ENDORSES THE INFORMATION THE ORGANIZATION OR WEBSITE MAY PROVIDE OR REC- OMMENDATIONS IT MAY MAKE. FURTHER, READERS SHOULD BE AWARE THAT INTERNET WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRIT- TEN AND WHEN IT IS READ.

For general information on our other products and services please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

**Trademarks:** Wiley, the Wiley logo, Wrox, the Wrox logo, Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be avail- able in electronic books.

*For our kids; you really are the best!*

##### About the Authors

**Adrian and Kathie Kingsley-Hughes** have written several successful technical/PC books on a variety of computer and IT-related topics. They have also developed numerous successful training manuals and Internet-based courses for nearly a decade.

Along with their day-to-day work, they currently teach online courses for several training providers, and Adrian also teaches several highly successful online courses for Barnes and Noble University. They have produced courses and materials that have been used extensively by many Fortune 500 companies and leading universities.

Put simply, they’re both geeks!

**Acquisitions Editor**

Katie Mohr

**Senior Development Editor**

Tom Dinse

**Technical Editor**

Andrew Moore

**Production Editor**

William A. Barton

**Copy Editor**

C.M. Jones

**Editorial Manager**

Mary Beth Wakefield

**Production Manager**

Tim Tate

##### Credits

**Graphics and Production Specialists**

Carrie A. Foster Heather Ryan Alicia B. South

**Quality Control Technician**

Dwight Ramsey

**Project Coordinator**

Ryan Steffen

**Media Development Specialists**

Angela Denny Kit Malone Travis Silvers

**Proofreading and Indexing**

Techbooks

**Vice President and Executive Group Publisher**

Richard Swadley

**Vice President and Executive Publisher**

Joseph B. Wikert

### Acknowledgments

A book like this is never the work of just the authors; it comes about as a result of a lot of hard work and the collaboration of dozens of people. The names on the cover represent just a small part of the equation (as authors, we feel that we are standing on the shoulders of a great many people who don’t get their names on the cover).

Knowing where to start thanking people can be difficult, but with this book it’s not. First and foremost, our thanks and appreciation go out to Katie Mohr, our tireless, hard-working acquisitions editor at Wiley, who first approached us with the opportunity to write this book. The amount of work and effort you put into this book, Katie, was just amazing, and the final product is infinitely better thanks to your input.

Our thanks also goes to our excellent development editor, Tom Dinse, who suggested a number of improvements and changes. Your feedback was very valuable, and it was a real pleasure to work with you!

There are a whole bunch of folks over at Wiley whom we haven’t mentioned—people who have worked anonymously in the background, laying out the book, indexing, proofreading, advertising, signing checks—we appreciate your valuable contribution to this title.

No electrons were harmed in the making of this book, but some did have to work extra hard in order to meet deadlines.

### Contents

[Acknowledgments ix](#_TOC_250383)

[Introduction xxiii](#_TOC_250382)

[How This Book Is Different xxiii](#_TOC_250381)

[Who This Book Is For xxiii](#_TOC_250380)

[How This Book Is Structured xxiv](#_TOC_250379)

[How to Tackle the Chapters xxv](#_TOC_250378)

[A Few Tips . . . xxv](#_TOC_250377)

[Conventions xxvi](#_TOC_250376)

[Source Code xxvi](#_TOC_250375)

[Errata xxvi](#_TOC_250374)

[p2p.wrox.com xxvii](#_TOC_250373)

Chapter 1: What is C#? 1

[The Name 1](#_TOC_250372)

[C# Overview 1](#_TOC_250371)

[History 2](#_TOC_250370)

[C# and CLR 2](#_TOC_250369)

[Diversions Into .NET 2](#_TOC_250368)

[Standards 3](#_TOC_250367)

[Other Implementations 3](#_TOC_250366)

[Sample C# Code 3](#_TOC_250365)

[The Benefits of Learning C# 5](#_TOC_250364)

[Summary 6](#_TOC_250363)

Chapter 2: Getting Started with C# 7

[Getting Into C# is Cheaper Than You Think! 7](#_TOC_250362)

[The Cheap End of the Spectrum 7](#_TOC_250361)

[How to Leverage Free C# Tools 10](#_TOC_250360)

[One Cheap Utility That Makes Life Easier! 13](#_TOC_250359)

[Alternative Text Editors and C# Tools 15](#_TOC_250358)

[Enterprise Tools - Visual Studio and Visual C# 15](#_TOC_250357)

[Summary 16](#_TOC_250356)

Chapter 3: Overview of C# 17

[C# 17](#_TOC_250355)

[C# Basics 17](#_TOC_250354)

[Examining the C# Source Code 18](#_TOC_250353)

[Types 19](#_TOC_250352)

[Value Types 19](#_TOC_250351)

[Reference Types 19](#_TOC_250350)

[Predefined Types 19](#_TOC_250349)

[Overloading 22](#_TOC_250348)

[Conversions 22](#_TOC_250347)

[Array Types 22](#_TOC_250346)

[Variables and Parameters 23](#_TOC_250345)

[Expressions 24](#_TOC_250344)

[Statements 26](#_TOC_250343)

[Classes 27](#_TOC_250342)

[Constants 28](#_TOC_250341)

[Fields 28](#_TOC_250340)

[Methods 28](#_TOC_250339)

[Properties 28](#_TOC_250338)

[Events 29](#_TOC_250337)

[Operators 29](#_TOC_250336)

[Indexers 29](#_TOC_250335)

[Instance Constructors 29](#_TOC_250334)

[Finalizers 29](#_TOC_250333)

[Static Constructors 30](#_TOC_250332)

[Inheritance 30](#_TOC_250331)

[Static Classes 30](#_TOC_250330)

[Structs 30](#_TOC_250329)

[Interfaces 30](#_TOC_250328)

[Delegates 31](#_TOC_250327)

[Enums 31](#_TOC_250326)

[Generics 31](#_TOC_250325)

[Iterators 32](#_TOC_250324)

[Nullable Types 32](#_TOC_250323)

[Summary 32](#_TOC_250322)

Chapter 4: C# Language Structure 35

[C# Programs 35](#_TOC_250321)

[Grammars 37](#_TOC_250320)

[Grammar Ambiguities 37](#_TOC_250319)

[Lexical Analysis 39](#_TOC_250318)

[Summary 55](#_TOC_250317)

Chapter 5: C# Concepts 57

[Application Startup 57](#_TOC_250316)

[Application Termination 58](#_TOC_250315)

[C# Declarations 58](#_TOC_250314)

[Members 60](#_TOC_250313)

[Namespace Members 61](#_TOC_250312)

[Struct Members 61](#_TOC_250311)

[Enumeration Members 61](#_TOC_250310)

[Class Members 61](#_TOC_250309)

[Interface Members 62](#_TOC_250308)

[Array Members 62](#_TOC_250307)

[Delegate Members 62](#_TOC_250306)

[Member Access 62](#_TOC_250305)

[Declared Accessibility 62](#_TOC_250304)

[Signatures 63](#_TOC_250303)

[Index Signatures 63](#_TOC_250302)

[Instance Constructor Signatures 63](#_TOC_250301)

[Method Signatures 63](#_TOC_250300)

[Operator Signatures 64](#_TOC_250299)

[Signatures and Overloading 64](#_TOC_250298)

[Scope 64](#_TOC_250297)

[Namespace and Type Names 66](#_TOC_250296)

[Memory Management in C# 66](#_TOC_250295)

[Summary 67](#_TOC_250294)

Chapter 6: Types 69

[Three Types of Types 69](#_TOC_250293)

[The Difference Between Value and Reference Types 69](#_TOC_250292)

[The C# Type System 70](#_TOC_250291)

[Value Types 70](#_TOC_250290)

[System.ValueType 71](#_TOC_250289)

[Default Constructors 72](#_TOC_250288)

[Struct Types 72](#_TOC_250287)

[Simple Types 73](#_TOC_250286)

[Integral Type 74](#_TOC_250285)

[Using Types 76](#_TOC_250284)

[Floating-Point Types 76](#_TOC_250283)

[Decimal Types 77](#_TOC_250282)

[bool Type 77](#_TOC_250281)

[Enumeration Types 77](#_TOC_250280)

[Reference Types 78](#_TOC_250279)

[Class Types 79](#_TOC_250278)

[Object Type 79](#_TOC_250277)

[String Type 79](#_TOC_250276)

[Array Types 79](#_TOC_250275)

[Delegate Types 79](#_TOC_250274)

[The null Type 79](#_TOC_250273)

[Boxing and Unboxing 80](#_TOC_250272)

[Nullable Types 80](#_TOC_250271)

[Summary 81](#_TOC_250270)

Chapter 7: Variables 83

[What are Variables? 83](#_TOC_250269)

[Not all Variables Are Created Equally 83](#_TOC_250268)

[Categories of Variables 84](#_TOC_250267)

[Static Variables 85](#_TOC_250266)

[Array Elements 85](#_TOC_250265)

[Instance Variables 86](#_TOC_250264)

[Value Parameter 87](#_TOC_250263)

[Reference Parameters 87](#_TOC_250262)

[Output Parameters 88](#_TOC_250261)

[Local Variables 88](#_TOC_250260)

[Default Values 89](#_TOC_250259)

[Definite Assignment 89](#_TOC_250258)

[Initially Assigned Variables 90](#_TOC_250257)

[Initially Unassigned Variables 90](#_TOC_250256)

[Summary 98](#_TOC_250255)

Chapter 8: Conversions 99

[Implicit Conversions 99](#_TOC_250254)

[Identity Conversions 100](#_TOC_250253)

[Implicit Numeric Conversions 100](#_TOC_250252)

[Implicit Enumeration Conversions 101](#_TOC_250251)

[Implicit Reference Conversions 101](#_TOC_250250)

[Boxing Conversions 102](#_TOC_250249)

[Implicit Type Parameter Conversions 102](#_TOC_250248)

[Implicit Constant Expression Conversions 103](#_TOC_250247)

[User Defined Implicit Conversions 103](#_TOC_250246)

[Explicit Conversions 103](#_TOC_250245)

[Explicit Numeric Conversions 103](#_TOC_250244)

[Explicit Enumeration Conversions 105](#_TOC_250243)

[Explicit Reference Conversions 106](#_TOC_250242)

[Unboxing Conversions 107](#_TOC_250241)

[Explicit Type Parameter Conversions 107](#_TOC_250240)

[User-Defined Explicit Conversions 107](#_TOC_250239)

[Standard Conversions 107](#_TOC_250238)

[Standard Implicit Conversions 107](#_TOC_250237)

[Standard Explicit Conversions 108](#_TOC_250236)

[User-Defined Conversions 108](#_TOC_250235)

[Anonymous Method Conversions 109](#_TOC_250234)

[Method Group Conversions 109](#_TOC_250233)

[Null Type Conversions 109](#_TOC_250232)

[Nullable Conversions 109](#_TOC_250231)

[Summary 110](#_TOC_250230)

Chapter 9: Expressions 111

[Classifications of Expressions 111](#_TOC_250229)

[Results of an Expression 112](#_TOC_250228)

[Expression Values 112](#_TOC_250227)

[Expressions and Operators 112](#_TOC_250226)

[Three Kinds of Operator 113](#_TOC_250225)

[Operator Precedence and Associativity 113](#_TOC_250224)

[Operator Overloading 115](#_TOC_250223)

[Lifted Operators 118](#_TOC_250222)

[Member Lookup 119](#_TOC_250221)

[Base Types 120](#_TOC_250220)

[Function Members 121](#_TOC_250219)

[Primary Expressions 125](#_TOC_250218)

[Literals 125](#_TOC_250217)

[Simple Names 126](#_TOC_250216)

[Parenthesized Expressions 126](#_TOC_250215)

[Member Access 126](#_TOC_250214)

[Invocation Expressions 127](#_TOC_250213)

[Element Access 127](#_TOC_250212)

[Default Value Expression 130](#_TOC_250211)

[Anonymous Methods 131](#_TOC_250210)

[Unary Expressions 131](#_TOC_250209)

[Cast Expressions 131](#_TOC_250208)

[Arithmetic Operators 131](#_TOC_250207)

[Shift Operators 132](#_TOC_250206)

[Relational/Type Testing Operators 132](#_TOC_250205)

[Logical Operators 133](#_TOC_250204)

[Conditional Logical Operators 133](#_TOC_250203)

[Null Coalescing Operator 134](#_TOC_250202)

[Assignment Operators 135](#_TOC_250201)

[Expression 135](#_TOC_250200)

[Constant Expressions 135](#_TOC_250199)

[Boolean Expressions 138](#_TOC_250198)

[Summary 138](#_TOC_250197)

Chapter 10: Statements 139

[What are Statements? 139](#_TOC_250196)

[C# Statements 141](#_TOC_250195)

[End Point and Reachability 142](#_TOC_250194)

[End Point 142](#_TOC_250193)

[Reachability 142](#_TOC_250192)

[Code Blocks 144](#_TOC_250191)

[Statement Lists 144](#_TOC_250190)

[Empty Statements 145](#_TOC_250189)

[Labeled Statements 145](#_TOC_250188)

[Declaration Statements 146](#_TOC_250187)

[Local Variable Declarations 146](#_TOC_250186)

[Local Constant Declarations 147](#_TOC_250185)

[Expression Statements 148](#_TOC_250184)

[Selection Statements 148](#_TOC_250183)

[Iteration Statements 154](#_TOC_250182)

[Jump Statements 156](#_TOC_250181)

[The using Statement 158](#_TOC_250180)

[The yield Statement 159](#_TOC_250179)

[Summary 160](#_TOC_250178)

Chapter 11: Namespaces 161

[What are Namespaces? 161](#_TOC_250177)

[Organizing Classes 161](#_TOC_250176)

[Controlling Scope 162](#_TOC_250175)

[Compilation Units 162](#_TOC_250174)

[Namespace Declarations 163](#_TOC_250173)

[Extern Alias Directives 164](#_TOC_250172)

[Using Directives 165](#_TOC_250171)

[Namespace Members 166](#_TOC_250170)

[Type Declarations 166](#_TOC_250169)

[Qualified Alias Member 167](#_TOC_250168)

[Summary 168](#_TOC_250167)

Chapter 12: Classes 169

[What are Classes? 169](#_TOC_250166)

[Class Declarations 169](#_TOC_250165)

[Class Modifiers 170](#_TOC_250164)

[Class Base Specification 171](#_TOC_250163)

[Base Classes 171](#_TOC_250162)

[Interface Implementations 171](#_TOC_250161)

[Class Body 171](#_TOC_250160)

[Partial Declarations 172](#_TOC_250159)

[Class Members 172](#_TOC_250158)

[Inheritance 173](#_TOC_250157)

[new Modifier 174](#_TOC_250156)

[Access Modifiers 174](#_TOC_250155)

[Static/Instance Members 174](#_TOC_250154)

[Constants 175](#_TOC_250153)

[Fields 176](#_TOC_250152)

[Static and Instance Fields 177](#_TOC_250151)

[readonly Fields 177](#_TOC_250150)

[Methods 178](#_TOC_250149)

[Method Parameters 179](#_TOC_250148)

[Static/Instance Methods 180](#_TOC_250147)

[Virtual Methods 180](#_TOC_250146)

[Override Method 180](#_TOC_250145)

[Sealed Methods 181](#_TOC_250144)

[Abstract Methods 181](#_TOC_250143)

[Method Body 181](#_TOC_250142)

[Properties 181](#_TOC_250141)

[Static/Instance Properties 182](#_TOC_250140)

[Accessors 182](#_TOC_250139)

[Virtual, Sealed, Override, and Abstract Accessors 183](#_TOC_250138)

[Events 184](#_TOC_250137)

[Field-Like Events 185](#_TOC_250136)

[Static/Instance Events 185](#_TOC_250135)

[Virtual, Sealed, Override, and Abstract Accessors 185](#_TOC_250134)

[Indexers 186](#_TOC_250133)

[Operators 187](#_TOC_250132)

[Unary Operators 189](#_TOC_250131)

[Binary Operators 189](#_TOC_250130)

[Conversion Operators 190](#_TOC_250129)

[Instance Constructors 190](#_TOC_250128)

[Static Constructors 191](#_TOC_250127)

[Finalizers 191](#_TOC_250126)

[Summary 192](#_TOC_250125)

Chapter 13: Structs 193

[What are Structs? 193](#_TOC_250124)

[Struct Declarations 194](#_TOC_250123)

[Struct Modifiers 195](#_TOC_250122)

[Struct Interfaces 195](#_TOC_250121)

[Struct Body 195](#_TOC_250120)

[Struct Members 195](#_TOC_250119)

[Differences Between Class and Struct 196](#_TOC_250118)

[Value Semantics 197](#_TOC_250117)

[Inheritance 197](#_TOC_250116)

[Assignments 198](#_TOC_250115)

[Default Values 198](#_TOC_250114)

[Boxing/Unboxing 198](#_TOC_250113)

[this 198](#_TOC_250112)

[Field Initializers 199](#_TOC_250111)

[Constructors 199](#_TOC_250110)

[Finalizers 199](#_TOC_250109)

[Static Constructors 199](#_TOC_250108)

[When to Use Structs 199](#_TOC_250107)

[Summary 200](#_TOC_250106)

Chapter 14: Arrays 201

[What is an Array? 201](#_TOC_250105)

[Array Types 203](#_TOC_250104)

[System.Array Type 204](#_TOC_250103)

[Creating Arrays 205](#_TOC_250102)

[Accessing Array Elements 205](#_TOC_250101)

[Array Members 205](#_TOC_250100)

[Array Covariance 205](#_TOC_250099)

[Array Initializers 206](#_TOC_250098)

[Summary 208](#_TOC_250097)

Chapter 15: Interfaces 209

[What is an Interface? 209](#_TOC_250096)

[Defining an Interface 210](#_TOC_250095)

[Interface Declarations 210](#_TOC_250094)

[Modifiers 211](#_TOC_250093)

[Explicit Base Interfaces 211](#_TOC_250092)

[Interface Body 212](#_TOC_250091)

[Interface Members 212](#_TOC_250090)

[Interface Methods 212](#_TOC_250089)

[Interface Properties 212](#_TOC_250088)

[Interface Events 213](#_TOC_250087)

[Summary 213](#_TOC_250086)

Chapter 16: Enums 215

[Enum Declarations 216](#_TOC_250085)

[Enum Modifiers 217](#_TOC_250084)

[Enum Members 218](#_TOC_250083)

[Beware Circular References 219](#_TOC_250082)

[System.Enum 219](#_TOC_250081)

[Enum Values and Operations 219](#_TOC_250080)

[Summary 220](#_TOC_250079)

Chapter 17: Delegates 221

[Delegates in Action 221](#_TOC_250078)

[Delegate Declarations 222](#_TOC_250077)

[Modifiers 222](#_TOC_250076)

[Declaring Delegates 223](#_TOC_250075)

[Invocation List 223](#_TOC_250074)

[Delegate Instantiation 224](#_TOC_250073)

[Summary 225](#_TOC_250072)

Chapter 18: Exceptions 227

[Throwing Exceptions 227](#_TOC_250071)

[System.Exception 228](#_TOC_250070)

[Common Exception Classes 228](#_TOC_250069)

[Handling Exceptions 229](#_TOC_250068)

[What If No Catch Clause Is Found? 229](#_TOC_250067)

[Summary 229](#_TOC_250066)

Chapter 19: Attributes 231

[Introduction to Attributes 231](#_TOC_250065)

[Attribute Classes 231](#_TOC_250064)

[Positional vs. Named Parameters 232](#_TOC_250063)

[Attribute Usage 232](#_TOC_250062)

[Types of Attribute Parameters 233](#_TOC_250061)

[Attribute Specification 233](#_TOC_250060)

[Attribute Instances 236](#_TOC_250059)

[Attribute Compilation 237](#_TOC_250058)

[Runtime Retrieval of Attribute Instances 237](#_TOC_250057)

[Reserved Attributes 238](#_TOC_250056)

[The Conditional Attribute 238](#_TOC_250055)

[Summary 240](#_TOC_250054)

Chapter 20: Generics 241

[C# Generics vs. C++ Templates 241](#_TOC_250053)

[Advantages of Generics 242](#_TOC_250052)

[Generic Class Declarations 242](#_TOC_250051)

[Type Parameters 243](#_TOC_250050)

[Type Parameter Differences 244](#_TOC_250049)

[Instance Type 244](#_TOC_250048)

[Generic Class Members 245](#_TOC_250047)

[Static Fields in Generic Classes 246](#_TOC_250046)

[Static Constructors in Generic Classes 246](#_TOC_250045)

[Access to Protected Members 246](#_TOC_250044)

[Overloading in Generic Classes 246](#_TOC_250043)

[Operators in Generic Classes 247](#_TOC_250042)

[Generic Struct Declarations 247](#_TOC_250041)

[Generic Interface Declarations 247](#_TOC_250040)

[Explicit Interface Member Implementations 248](#_TOC_250039)

[Generic Delegate Declarations 248](#_TOC_250038)

[Constructed Types 249](#_TOC_250037)

[Type Arguments 249](#_TOC_250036)

[Open and Closed Types 249](#_TOC_250035)

[Members of Constructed Types 250](#_TOC_250034)

[Using Alias Directives 250](#_TOC_250033)

[Generic Methods 250](#_TOC_250032)

[Where Generics Aren’t Used 252](#_TOC_250031)

[Constraints 253](#_TOC_250030)

[Summary 256](#_TOC_250029)

Chapter 21: Iterators 257

[Iterator Block 258](#_TOC_250028)

[Iterator Blocks and Compile-time Errors 259](#_TOC_250027)

[Enumerator Interfaces 259](#_TOC_250026)

[Enumerable Interfaces 259](#_TOC_250025)

[Yield Type 260](#_TOC_250024)

[This 260](#_TOC_250023)

[Enumerator Objects 260](#_TOC_250022)

[The MoveNext Method 260](#_TOC_250021)

[Execution Interruptions 261](#_TOC_250020)

[The Current Property 262](#_TOC_250019)

[The Dispose Method 262](#_TOC_250018)

[Enumerable Objects 263](#_TOC_250017)

[GetEnumerator Method 263](#_TOC_250016)

[Summary 264](#_TOC_250015)

Chapter 22: Unsafe Code 265

[What is Unsafe Code? 265](#_TOC_250014)

[Advantages and Disadvantages of Unsafe Code 266](#_TOC_250013)

[Advantages of Unsafe Code 266](#_TOC_250012)

[Disadvantages of Unsafe Code 266](#_TOC_250011)

[Unsafe Code Contexts 266](#_TOC_250010)

[Pointer Basics 268](#_TOC_250009)

[Void Pointers 268](#_TOC_250008)

[Pointer Operators 268](#_TOC_250007)

[Unsafe in Action 269](#_TOC_250006)

[Using the fixed Modifier 270](#_TOC_250005)

[sizeof Operator 272](#_TOC_250004)

[Using stackalloc 273](#_TOC_250003)

[Compiling Unsafe Code 273](#_TOC_250002)

[Summary 273](#_TOC_250001)

Appendix A: C# Grammar 275

Appendix B: Naming Conventions 337

Appendix C: Standard Library 345

Appendix D: Portability 359

Appendix E: XML Documentation Comments 363

[Index 367](#_TOC_250000)

### Introduction

In this book, we’re going to take a very detailed walk through the entire C# programming language. This book is not a “learn C# in five minutes” manual, nor is it a book that looks at how to build a couple of applications that you will probably never need to know how to build, because they have no relation to your job or your hobby. That kind of book can give you only the very simplest of overviews of a pro- gramming language.

##### How This Book Is Different

This book is different. Instead of giving you a basic overview of the language as many other books do (think of them as a bit like viewing a globe of the Earth, offering the outlines of the continents and coun- tries and a few basic features like lakes and so on but not much in the way of detail), this book takes you all the way into the language and looks at what makes it tick (going back to the map analogy, you will zoom in from Earth orbit right down to street level, where you’ll be able to see every street name and all the buildings).

All this doesn’t mean that we’re not going to spend some time looking at the bigger C# picture. We’re going to spend a few chapters looking at broader topics of C# (such as looking at what C# is and how to get started with C# before taking in an overview of the C# language). These foundation chapters will allow you to orient yourself before delving into the detailed look at the various aspects of the language.

After giving you a few foundation chapters, we then dive right into C#. We start off by looking at the structure and concepts of the C# language. We then take a close look at C# types, variables, and conver- sions. Building on these chapters, we then progress onto examining the syntax of C# expressions and statements, before moving on to looking at how C# uses namespaces.

We then move on to look at C# classes, structs, arrays, and enums and then delegates, exceptions, attributes, generics, and iterators. We round off the main chapters by taking a look at safe and unsafe coding practices in C#. The book ends with a number of appendixes that detail the C# grammar, naming conventions, portability, and XML documentation comments.

##### Who This Book Is For

This book isn’t designed to teach C#. It’s designed to aid those who already have a basic understanding of C# to be able to take the skills that they have and build on them by leveraging more advanced tech- niques and aspects of the language.

If you don’t have any experience at all with C#, we suggest that you take a look at the range of Wrox titles and choose a beginner-level book. This will give you all the basic knowledge you need to be able to take advantage of the advanced techniques.

##### How This Book Is Structured

We start with the Introduction (what you’re reading now!). Following is the rest of the book:

* **Chapter 1: What is C#?** This chapter takes a look at what C# is, its origins, and its history.
* **Chapter 2: Getting Started with C#.** You don’t need a lot of software to get started program- ming with C#. In this chapter we look at what you really need and a few things that will make your life a little easier.
* **Chapter 3: Overview of C#.** Here we give you a whirlwind tour of C# and highlight some of the most important features of this powerful and flexible programming language.
* **Chapter 4: C# Language Structure.** In this chapter we take a look at the structure of the C# lan- guage, paying special attention to the lexical and syntactic grammar, the tokens, and the directives.
* **Chapter 5: C# Concepts.** In this chapter we take a look at many of the key concepts in C#, such as application startup and termination, members and member access, and overloading.
* **Chapter 6: Types.** We now begin to look at specific aspects of the C# language, starting with types. We look at value types and reference types and boxing and unboxing.
* **Chapter 7: Variables.** Next we look at a topic that is at the heart of data manipulation: variables.
* **Chapter 8: Conversions.** This chapter takes a look at conversion in C# (in particular, implicit, explicit, and standard conversions).
* **Chapter 9: Expressions.** At the heart of C# coding are expressions. In this chapter we take a look at the variety of expressions available in C#.
* **Chapter 10: Statements.** Lines of code are known as statements. In this chapter we look at the structure of statements and examine a number of different statements.
* **Chapter 11: Namespaces.** This chapter takes a look at how C# utilizes namespaces, which allows for disambiguation of items having the same name.
* **Chapter 12: Classes.** In this chapter we examine classes and how they are used to compartmen- talize code in C#.
* **Chapter 13: Structs.** In this chapter we look at structs and how to use them in your coding.
* **Chapter 14: Arrays.** Arrays are a great way to structure data to make it easier to access and work with. In this chapter we look at the different sorts of arrays available in C#.
* **Chapter 15: Interfaces.** In this chapter we examine interfaces in C# and look at declarations, members, qualified member names, and implementations.
* **Chapter 16: Enums.** Enums are strongly typed constants, and in this chapter we examine how they are used in C# coding.
* **Chapter 17: Delegates.** This chapter looks at delegate declarations, instantiations, and invoca- tions in C#.
* **Chapter 18: Exceptions.** This chapter examines exceptions and looks at their causes, handling, and exception classes.
* **Chapter 19: Attributes.** This chapter looks at attribute classes, instances, and reserved attributes.
  + **Chapter 20: Generics.** Generics are a new and interesting feature in C#. In this chapter we take a look at how to leverage generic declarations.
  + **Chapter 21: Iterators.** Iterators allow for core-concise and faster code to be written. In this chap- ter we examine a number of different iterators available in C#.
  + **Chapter 22: Safe and Unsafe Code.** In this chapter we look at how to make use of unsafe code features in C# without compromising the rest of the project.
  + **Appendix A: C# Grammar.**
  + **Appendix B: Naming Conventions.**
  + **Appendix C: Standard Library.**
  + **Appendix D: Portability.**
  + **Appendix E: XML Documentation Comments.**

##### How to Tackle the Chapters

How you work your way through this book is entirely up to you. If you are relatively new to C#, you’ll probably want to start off right at the beginning and read Chapters 1 through 10. Then you can dip in and out of the other chapters as you see the need and as your programming skills with C# improve. If you are already a C# user, this book is likely to be more of a reference for you rather than a book that you read beginning to end, and you can dig into the various chapters as you need the information.

The appendixes are resources for you to dip into when you need information on a particular aspect of C#. Unless you are totally committed to C#, we don’t expect you to read these beginning to end. (Feel free to do so if you want to — just remember that we warned you!)

##### A Few Tips . . .

This is a pretty big book and as such may seem daunting. As we sit at our desks writing this book, we can look up at the shelves in the office and see a number of big, thick books that we haven’t looked at in ages. We don’t want this book to be one that just sits on the shelf gathering dust. We suggest that you make the book as readable as possible. As you read it and find something that’s of particular use, get a highlighter pen (or better still, a fine colored pen, since that gives you better control than a highlighter) and highlight it. Additionally, make notes in the margin as to why you found that bit interesting, useful, or relevant. By doing so when you are reading a given page, it will make the information easier to find the next time you want to refer to it.

Also, as you are reading, you might find it useful to turn down the corners of pages or add your own notes using Post-it Notes. Some of the most useful books we have on our shelves are ones that we’ve personalized in this way.

You will also need access to a Windows-based PC with the Microsoft .NET Framework installed on it (chances are that you already have this installed). You will also need to have a minimum of a basic Windows text editor and a working knowledge of using Windows command-line applications.

##### Conventions

To help you get the most from the text and keep track of what’s happening, we’ve used a number of con- ventions throughout the book.

*Tips, hints, tricks, and asides to the current discussion are offset and placed in italics like this.*

As for styles in the text:

* New terms and important words are *highlighted* when they’re introduced.
* Keyboard combinations appear like this: Ctrl+A.
* Filenames, URLs, and code within the text appear in monospaced font, like this:

persistence.properties.

* Code is presented in two ways:

A gray background highlights examples of new and important code.

The gray highlighting is not used for code that’s less important in the present context or that has been shown before.

##### Source Code

As you work through the examples in this book, you may choose either to type all the code manually or to use the source code files that accompany the book. All of the source code used in this book is available for download at [http://www.wrox.com.](http://www.wrox.com/) At the site, simply locate the book’s title (either by using the Search box or by using one of the title lists) and click the Download Code link on the book’s detail page to obtain all the source code for the book.

*Because many books have similar titles, you may find it easiest to search by ISBN; this book’s ISBN is 0-470-04641-4 (changing to 978-0-470-04641-8 as the new industry-wide 13-digit ISBN numbering system is phased in by January 2007).*

Decompress the downloaded code with your favorite compression tool. Alternatively, you can go to the main Wrox code-download page at <http://www.wrox.com/dynamic/books/download.aspx>to see the code available for this book and for all other Wrox books.

##### Errata

We make every effort to ensure that there are no errors in the text or in the code. However, no one is per- fect, and mistakes do occur. If you find an error in one of our books, like a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata, you may save another reader hours of frustration, and at the same time you will be helping us provide even higher-quality information.

To find the errata page for this book, go to [http://www.wrox.com](http://www.wrox.com/) and locate the title using the Search box or one of the title lists. Then, on the book details page, click the Book Errata link. On this page you can view all errata that has been submitted for this book and posted by Wrox editors. A complete book list including links to each book’s errata is also available at [www.wrox.com/misc-pages/booklist.shtml.](http://www.wrox.com/misc-pages/booklist.shtml)

If you don’t spot “your” error on the Book Errata page, go to [www.wrox.com/contact/techsupport](http://www.wrox.com/contact/techsupport)

.shtml and complete the form there to send us the error you have found. We’ll check the information and, if appropriate, post a message to the book’s errata page and fix the problem in subsequent editions of the book.

##### p2p.wrox.com

For author and peer discussion, join the P2P forums at p2p.wrox.com. The forums are a Web-based sys- tem for you to post messages related to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

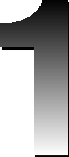
At [http://p2p.wrox.com](http://p2p.wrox.com/) you will find a number of different forums that will help you not only as you read this book but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to p2p.wrox.com and click the Register link.
2. Read the terms of use and click Agree.
3. Complete the required information to join as well as any optional information you want to provide and click Submit.
4. You will receive an e-mail with information describing how to verify your account and complete the joining process.

*You can read messages in the forums without joining P2P, but in order to post your own messages, you must join.*

Once you join, you can post new messages and respond to messages other users post. You can read mes- sages at any time on the Web. If you would like to have new messages from a particular forum e-mailed to you, click the Subscribe to this Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to ques- tions about how the forum software works as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.











### C# Concepts

In this chapter you examine some basic concepts in C#. The purpose of this analysis is to get you up to speed on the terminology and ideas that we will be expanding on later in the book. This chapter is worth a quick read even if you’re familiar with, say, C++ or Java.

##### Application Startup

Let’s begin by looking at how application startup works in C#.

An application starts to run when the execution environment calls a designated method, called the

*entry point*. This entry point is always called Main. The entry point can take on one of four signatures:

* + static void Main() {...}
  + static void Main(string[] args) {...}
  + static int Main() {...}
  + static int Main(string[] args) {...}

As you can see, it is possible for the entry point to return an int value that can be used during application termination.

It is possible for the entry point to have one and only one parameter. This parameter can be called anything, but it has to conform to the following rules:

* + The value of the parameter cannot be null.
  + If you call the parameter args and if the length of the array designated by args is greater than zero, the array members args[0] through args[args.Length-1], inclusive, will be strings called *application parameters*. These are supplied with implementation-defined values by the host environment prior to the application being started (think of command- line arguments).

There are also a few simple rules related to the Main method:

* A program can only contain one Main method entry point. Multiple definitions through over- loading are not allowed.
* The entry point cannot be a generic class declaration or a generic struct declaration.

##### Application Termination

You’ve looked at application startup; now you’ll look at application termination.

Application termination is where control is returned to the execution environment. If the return type of the application’s entry point method is set to int, the value returned will be the application’s termina- tion status code. This code allows the execution environment to determine whether the termination was successful or not.

If the return type of the entry point method is void, reaching the right closing brace (}), which ends the method, or executing a return statement that has no expression will both result in a termination status code of 0.

At the point just before an application termination, finalizers (see Chapter 3) for all of the objects used that have not yet been dealt with by garbage collection are called (unless this is suppressed).

##### C# Declarations

Declarations in C# are used to define separate aspects of a C# program. C# programs are built around a number of declarations:

* **Type declarations.** Used to define classes, delegates, enums, interfaces, and structs
* **Namespace declarations.** Contain type declarations and nested namespace declarations
* **Various other declarations.** For example, class declarations, which can contain declarations such as:
  + Constants
  + Events
  + Fields
  + Finalizers
  + Indexers
  + Instance constructors
  + Methods
  + Nested types
    - Operators
    - Properties
    - Static constructors

A declaration defines a name in the declaration space to which the declaration belongs. A compiler error will be generated if two or more declarations introduce members with the same name in a declaration space, unless:

* Two or more namespace declarations with the same name are allowable in the same declaration space. When this is the case, the individual namespace declarations are combined to form a sin- gle logical namespace with a single declaration space.
* A namespace declaration and one or more type declarations in the same declaration space can have the same name as long as the type declarations all have a minimum of one type parameter.
* Two or more methods with the same name but with different signatures are allowed in the same declaration.
* Two or more type declarations with the same name but different numbers of type parameters are allowed in the same declaration space.
* Two or more type declarations with the partial modifier in the same declaration space can have the same name, the same number of type parameters, and the same classification. These are combined into a single declaration space.

Declarations in separate programs but in the same namespace declaration space are allowed to have the same name.

*A type declaration space can never contain different kinds of members that have an identical name.*

There are a number of different kinds of namespace declarations:

* Within the source files of a program, namespace-member-declarations with no enclosing namespace-declaration are members of a single combined declaration space called the global declaration space.
* Within the source files of a program, namespace-member-declarations within namespace- declarations that have the same fully qualified namespace name are members of a single combined declaration space.
* Each compilation-unit and namespace-body has an alias declaration space. The extern- alias-directive and using-alias-directive of the compilation-unit or namespace- body contributes a member to the alias declaration space.
* Each nonpartial class, struct, or interface declaration creates a new declaration space. Each partial class, struct, or interface declaration contributes to a declaration space shared by all matching parts in the same program. All the names are introduced into this declaration space through the type-parameter-list and class-member-declarations, struct-member-declarations, or interface-member-declarations. With the exception of overloaded instance constructor declarations and static constructor declarations, a class or struct member declaration are not able to introduce a member by the same name as the class or struct. A class, struct, or interface permits the declaration of overloaded methods and indexers. Also, a class or struct permits the declara- tion of overloaded instance constructors, operators, and types.
* Each enumeration declaration creates a new declaration space. The names are introduced into the declaration space through enum-member-declarations.
* Every block or switch block creates a declaration space for local variables and local constants called the local variable declaration space. Names are introduced into this declaration space through local-variable-declarations and local-constant-declarations.
* Every block or switch block creates a separate declaration space for labels called the label decla- ration space of the block. All names are introduced into this declaration space through labeled-statements, and the names are referenced through goto-statements.

The order in which the names are declared is usually of no significance. For example, the order is not significant for the declaration and use of:

* Constants
* Events
* Finalizers
* Indexers
* Instance constructors
* Methods
* Namespaces
* Operators
* Properties
* Static constructors
* Types

However, declaration order is significant in the following circumstances:

* Declaration order for field declarations and local variable declarations determines the order in which any initializers are executed.
* Local variables and local constants have to be defined before they are used.

Declaration order for enum member declarations is important when constant-expression values are not present.

##### Members

Namespaces and types all have members. Members of a type can either be declared in the type or inher- ited from the base class of the type.

When a type inherits from a base class, all members of the base class (except finalizers, instance con- structors, and static constructors) become members of the derived type.

The declared accessibility of a base class member does not control whether the member’s inherited- inheritance covers any member that isn’t an instance constructor, static constructor, or finalizer.

Namespace Members

Any namespaces and types that don’t have an enclosing namespace are members of the global namespace. Any namespaces and types declared within a namespace are members of that namespace.

Namespaces have no access restrictions and are always publicly accessible. You cannot declare private, protected, or internal namespaces.

Struct Members

The members of a struct are the members declared in the struct and the members inherited from the direct base class of the struct System.ValueType and the indirect base class object.

Enumeration Members

The members of any enumeration are the constants declared in the enumeration itself and the members inherited from the direct base class System.Enum of the enumeration, along with the indirect base classes System.ValueType and object.

Class Members

The members of a class are the members declared in the class along with the members inherited from the base class.

The members inherited from the base class include all of the following of the base class:

* + Constants
  + Events
  + Fields
  + Indexers
  + Methods
  + Operators
  + Properties
  + Types

The following are not included:

* + Finalizers
  + Instance constructors
  + Static constructors

*Base class members are inherited irrespective of their accessibility.*

A class declaration can contain the following declarations:

* Constants
* Events
* Fields
* Finalizers
* Indexers
* Instance constructors
* Methods
* Operators
* Properties
* Static constructors
* Types

Interface Members

Members of an interface are the members declared in the interface along with those declared in the base interfaces of the interface.

Array Members

All the members of an array are inherited from class System.Array, which is the abstract base type of all array types.

Delegate Members

All the members of a delegate are inherited from class System.Delegate. Delegates will be covered in greater detail in later chapters.

##### Member Access

Member declarations are allowed control over member access using declared accessibility (covered in the following section). When access is allowed, the member is accessible; otherwise, it is inaccessible.

Declared Accessibility

Declared accessibility of a member can be set to one of the following five categories:

* **Public.** In this case, access is not limited.
* **Protected.** Access is limited to the containing class or type derived from the containing class.
  + **Internal.** Access is limited to the program.
  + **Protected internal.** Access is limited to the program or types derived from the containing class.
  + **Private.** Access is limited to the containing type.

When a member declaration does not include any access modifiers, there is a default declared accessibility:

* + Namespaces implicitly have public declared accessibility (in fact, no access modifiers are allowed on namespace declarations).
  + Types declared in compilation units or namespaces default to internal declared accessibility.
  + Class members default to private declared accessibility.
  + Struct members default to private declared accessibility.
  + Interface members implicitly have public declared accessibility (no access modifiers are allowed).
  + Enumeration members implicitly have public declared accessibility (no access modifiers are allowed).

##### Signatures

In C#, all indexes, instance constructors, methods, and operators are characterized by their signature. The following sections provide a rundown of the signature of each of these.

Index Signatures

The signature of an indexer is made up of the type of each of its formal parameters. They are processed in left-to-right order.

The signature of an indexer does not include the element type or parameter names. Additionally, it does not include the params modifier that can be specified for the right-most parameter.

Instance Constructor Signatures

The signature of an instance constructor is made up of the type and style of the parameters (that is, whether it is value, reference, or output). They are processed in left-to-right order.

The signature of an instance constructor does not include the parameter names or the params modifier specified for the right-most parameter.

Method Signatures

The signature of a method is made up of the following:

* + The name of the method
  + The number of type parameters
  + The type and style of the parameters (that is, whether it is value, reference, or output)

They are processed in left-to-right order.

Note that the signature of a method does not include the following:

* Return type
* Parameter names
* Type parameter names
* The params modifier that can be specified for the right-most parameter

Operator Signatures

The signature of an operator is made up of the name of the operator and the type of each of the parame- ters. They are processed in left-to-right order.

The signature of an operator does not include the following:

* Result type
* Parameter names

Signatures and Overloading

Signatures are a mechanism that allows for the overloading of members in classes, interfaces, and structs.

Overloading Indexers

Overloading indexers allows a class, interface, or struct to declare multiple indexers as long as their sig- natures are unique within that class, interface, or struct.

Overloading Instance Constructors

Overloading instance constructors allows a class or struct to declare multiple instance constructors as long as their signatures are different within that class or struct.

Overloading Methods

Overloading a method allows a class, interface, or struct to declare multiple methods where each has the same name as long as their signatures are different within the class, interface, or struct.

Overloading Operators

Overloading operators allows a class or struct to declare multiple operators with the same name as long as their signatures are different within that class or struct.

##### Scope

Scope is a term used in programming to describe the region of code within a program where it is possi- ble to refer to an entity that’s been declared without having to qualify the name.

It is possible for various scopes to be nested, and an inner scope can declare again the meaning of a name from an outer scope. In this case, the name from the outer scope is hidden in the region of code covered by the inner scope. Furthermore, access to the outer name is possible only by qualifying the name.

Here are the rules governing scope:

* + The scope of a namespace member declared by a namespace-member-declaration that has no enclosing namespace-declaration is the entire program.
  + The scope of a namespace member declared by a namespace-member-declaration within a namespace-declaration that has the fully qualified name is N (a shorthand representation) is the namespace-body of every namespace-declaration that has the fully qualified name is N or starts with N and is followed by a period.
  + The scope of a namespace member declared by a namespace-member-declaration that has no enclosing namespace-declaration is the entire program.
  + The scope of a namespace member declared by a namespace-member-declaration within a namespace-declaration that has the fully qualified name is N is the namespace-body of every namespace-declaration that has the fully qualified name N or starts with N and is fol- lowed by a period.
  + The scope of a name defined by an extern-alias-directive covers the using-directives, global-attributes, and namespace-member-declarations of the compilation-unit or namespace-body where the extern-alias-directive is found.
  + The scope of a name defined by a using-directive covers the global-attributes and namespace-member-declarations of the compilation-unit or namespace-body in which the using-directive is found.
  + The scope of a member declared by a class-member-declaration is the class-body where the declaration is found. The scope of a class member also extends to the class-body of derived classes included in the accessibility domain of the member.
  + The scope of a member declared by a struct-member-declaration is the struct-body

where the declaration is found.

* + The scope of a member declared by an enum-member-declaration is the enum-body where the declaration is found.
  + The scope of a parameter declared in a method-declaration is the method-body of that

method-declaration.

* + The scope of a parameter declared in an indexer-declaration is the accessor-declarations

of that indexer-declaration.

* + The scope of a parameter declared in an operator-declaration is the block of that

operator-declaration.

* + The scope of a parameter declared in a constructor-declaration is the constructor- initializer and block of that constructor-declaration.
  + The scope of a label declared in a labeled-statement is the block in which the declaration occurs.
* The scope of a local variable declared in a local-variable-declaration is the block in which the declaration occurs.
* The scope of a local variable declared in a switch-block of a switch statement is the switch

block.

* The scope of a local variable declared in a for-initializer of a for statement is the

for-initializer, the for-condition, and the for-iterator, along with the contained statement of the for statement.

* The scope of a local constant declared in a local-constant-declaration is the block in which the declaration is found.

##### Namespace and Type Names

A number of contexts in a C# program require a namespace-name or a type-name to be specified. The following shows the syntax for namespaces and type names.

namespace-name:

namespace-or-type-name

type-name:

namespace-or-type-name

namespace-or-type-name:

identifier type-argument-listopt qualified-alias-member

namespace-or-type-name . identifier type-argument-listopt

The namespace-or-type-name of a namespace-name has to refer to a namespace. Type arguments cannot be in a namespace-name.

A type-name is a namespace-or-type-name that refers to a type. Following resolution as described in the following section, the namespace-or-type-name of a type-name has to refer to a type.

##### Memory Management in C#

C# has at its core a rigorous memory management scheme built into the .NET Framework. This means that programmers have to write less code. Automatic memory-management policies are carried out by the garbage collector, and these policies mean that the programmer doesn’t have to manually allocate and free memory used by objects.

Here is the general lifecycle of an object:

1. The object is created.
2. Memory is allocated for the object.
3. The constructor is run.
4. The object is now live.
5. If the object is no longer in use (other than running finalizers), it needs finalization.
6. Finalizers are run (unless overridden).
7. The object is now inaccessible and is available for the garbage collector to carry out clean-up.
8. The garbage collector frees up associated memory.

##### Summary

In this chapter you looked at a number of key concepts in C#.

* Application startup
* Application termination
* Declarations
* Members
* Member access
* Signatures
* Overloading
* Scope
* Namespaces and type names
* Memory management

In Chapter 6, you look at C# types.



### Types

Everything in C# is a type, so it’s important to get a handle on what these different types are and how they work within the confines of C#.

##### Three Types of Types

For the purposes of this chapter, there are three kinds of type in C#:

* Value types
* Reference types
* Type-parameter types (form part of generics and are discussed in Chapter 20)

There is also a fourth type, used only in unsafe code called pointers, which you will come across in Chapter 22.

The Difference Between Value and Reference Types

There is a fundamental difference between value and reference types that is quite easy to understand:

* **Value type variables:** These types directly contain data.
* **Reference type variables:** These types contain only a reference to data and are known as objects.

This fundamental difference leads to some very interesting possibilities. For example, with refer- ence types it’s possible for two or more variables to reference the same object, and if an operation is carried out on one variable, this affects the object referenced by all the other variables.

The situation is different with value types. With value types, the variables each have their own copy of data, and working on one copy does not affect any of the others. Thus:

* Reference types refer to a single source of data.
* Value types each have their own copy of data.

This fundamental difference has huge practical applications in programming but can also be the source of a lot of problems if you’re not aware of it.

ref and out Parameters

When a variable is either a ref or out parameter, it is important to note that the variable is in essence an alias for another variable rather than being a distinct variable itself. It doesn’t have its own storage but instead references the storage area of another variable.

##### The C# Type System

Every value of any type in C# is unified and can be treated as an object, and every type, either directly or indirectly, derives from the object class type. Also, object will be the base class for all types.

How the two types are treated as objects is also different:

* The values of reference types are handled as objects by simply viewing the value as object.
* The values of value types can only be treated as objects by carrying out boxing and unboxing operations (explained later in this chapter).

##### Value Types

Value types can be either:

* A struct type
* An enumeration type

C# offers a host of predefined struct types called simple types, and these are identified through reserved words, the syntax of which is listed as follows:

value-type:

struct-type enum-type

struct-type:

type-name simple-type nullable-type

simple-type:

numeric-type bool

numeric-type:

integral-type floating-point-type decimal

integral-type: sbyte byte short ushort int

uint long ulong char

floating-point-type: float

double

enum-type:

type-name

nullable-type:

non-nullable-value-type ?

non-nullable-value-type: enum-type

type-name simple-type

All value types will implicitly inherit from the class object, and it is not possible for types to derive a value type, which makes value types sealed.

One key aspect of a variable of the value types is that they will always, without exception, contain a value of that type. It is impossible for a value type to have a value that is null. Equally, the value of a value type cannot reference an object of a more derived type.

Assignment to any variable of a value type results in a copy of that value being assigned, keeping the original value safe from alteration. This is different from reference values, where the reference is copied but not the object itself.

System.ValueType

All value types inherit implicitly from the System.ValueType class. This class inherits from the object

class.

Bear in mind that the System.ValueType class is a class-type from which every value-type is derived rather than being a value-type itself.

Default Constructors

All value types implicitly declare a public parameterless instance constructor. This constructor is called a default constructor, and it returns a zero-initialized instance known as a default value for the type.

For all simple types, the default value will be produced by a bit pattern that corresponds to all zeros.

**Type Default value**

sbyte byte short ushort int uint long

ulong

0

Char \x0000

Float 0.0f

Double 0.0d

Decimal 0m

Bool false

For enum-types E (a shorthand notation), the default is 0.

For struct-type, the default value will be the value produced when setting all the value types to their default values and all reference fields to null.

Struct Types

A struct type is a value type that can declare any of the following:

* Constants
* Fields
* Indexers
* Instance constructors
* Methods
* Nested types
  + Operators
  + Properties
  + Static constructors

Simple Types

The predefined struct types in C# are called simple types. These are identified through the use of reserved words. These reserved words are aliases for predefined struct types contained in the System namespace.

Here is a list of reserved words, along with their aliased types:

**Reserved word Aliased type**

Ushort System.Uint16

Bool

System.Boolean

Byte

System.Byte

Char

System.Char

Decimal

System.Decimal

Double

System.Double

Float

System.Single

Int

System.Int32

Long

System.Int64

Sbyte

System.Sbyte

Short

System.Int16

Uint

System.Uint32

Ulong

System.Uint64

You can carry out more operations on simple types than is possible on other struct types:

* + Most simple types allow values to be created by writing literals.
  + When the operands of an expression are all value types (known as a constant expression), the compiler will evaluate the expression when it is compiled. This speeds program execution.
  + Constants of simple types can be declared using const declarations.

Integral Type

C# supports several different integral types, described in the following table:

|  |  |  |
| --- | --- | --- |
| **Type** | **Description** | **Value range** |
| Sbyte | Signed 8-bit integer | -128 |
|  |  | to |
|  |  | 127 |
| Byte | Unsigned 8-bit integer | 0 |
|  |  | to |
|  |  | 255 |
| Short | Signed 16-bit integer | -32768 |
|  |  | to |
|  |  | 32767 |
| Ushort | Unsigned 16-bit integer | 0 |
|  |  | to |
|  |  | 65535 |
| Int | Signed 32-bit integer | -2147483648 |
|  |  | to |
|  |  | 2147483647 |
| Uint | Unsigned 32-bit integer | 0 |
|  |  | to |
|  |  | 4294967295 |
| Long | Signed 64-bit integer | -9223372036854775808 |
|  |  | to |
|  |  | 9223372036854775807 |
| Ulong | Unsigned 64-bit integer | 0 |
|  |  | to |
|  |  | 18446744073709551615 |
| Char | Unsigned 16-bit integer | 0 |
|  | corresponding to the Unicode | to |
|  | character set | 65535 |

*Note that while* char *types are integral types, there are two differences:*

* *Implicit conversion to the* char *type from other types is not supported.*
* *Constants of the* char *type are written as* character-literals *or* integer-literals *and in combination with a cast to the* char *type.*

Types can also be signed (positive and negative) or unsigned:

**Type Signed?**

Sbyte

Byte Short Ushort Int Uint Long Ulong Char Float Double

Decimal

Bool

Yes

No Yes No Yes No Yes No N/A Yes Yes Yes No

Each type also occupies a specific number of bytes in memory.

|  |  |
| --- | --- |
| **Type** | **Bytes Occupied** |
| Sbyte | 1 |
| Byte | 1 |
| Short | 2 |
| Ushort | 2 |
| Int | 4 |
| Uint | 4 |
| Long | 8 |
| Ulong | 8 |
| Char | 2 |
| Float | 4 |
| Double | 8 |
| Decimal | 12 |
| Bool | 1/2 |

To reduce on the system requirements of code, use the most appropriate type for your data. For example, if a short integer will do instead of a long one, use it and save six bytes for each entry. Using decimal instead of short would mean that each variable would require 12 bytes instead of two.

The integral-type unary and binary operators always use the following levels of precision:

* signed 32-bit precision
* unsigned 32-bit precision
* signed 64-bit precision
* unsigned 64-bit precision

Using Types

Using types is easy. The type names prefix variable names. For example:

string str1 = “Hello, World!”;

string str2 = str1; //str1 equals str2 int x = 10;

int y = x; // y equals 10 y = 20; // y now equals 20

Floating-Point Types

C# supports two floating-point types:

* Float — Values ranging from approximately 1.5  1045 to 3.4  1038. Float has a precision accu- rate to 7 digits.
* Double — Values ranging from approximately 5.0  10324 to 1.7  10308. Double has a precision accurate to 15 or 16 digits.

float and double are represented using 32-bit single-precision and 64-bit double-precision formats. The following sets of values are allowed:

* **Positive and negative zero.** In most cases, these are identical to simple zero, but some opera- tions (division operations) distinguish between the two.
* **Positive and negative infinity.** Infinities are generated by dividing a nonzero number by zero.
* **Not-a-Number (NaN).** These are produced by invalid floating-point operations (carrying out a divide zero by zero, for example).

Floating-point operations do not produce exceptions. Instead, they produce one of the following in an exception situation:

* Zero
* Infinity
* NaN

Here are the rules by which these are generated:

* + The result of a floating-point operation can be rounded to the nearest value that can be repre- sented by the destination format, and this may cause a nonzero value to be rounded to zero.
  + If the magnitude of the result of a floating-point operation is too big for the destination format, the result of the operation is transformed into positive infinity or negative infinity.
  + If a floating-point operation is invalid, the result of the operation produces NaN.
  + If one or both operands of a floating-point operation are NaN, the result of the operation also becomes NaN.

Decimal Types

A decimal type is a 128-bit type. It has the range 1  10-28 to 1  1028 and has at least 28 significant digits. The decimal type is ideally suited for financial calculations.

If a decimal arithmetic operation produces a result where the magnitude is too large for the decimal for- mat, a System.OverflowException is thrown.

Again, be aware that rounding operations can cause a loss of precision or a rounding to zero.

bool Type

The bool type represents a Boolean logic quantity that can be either true or false. There is no stan- dard conversion between bool and other types, and it is distinct to integral types.

Enumeration Types

An enumeration type is a distinct type with named constants. Each enumeration type has an underlying type, which will be one of the following:

* + byte
  + sbyte
  + short
  + ushort
  + int
  + uint
  + long
  + ulong

Enumeration types are defined through enumeration declarations.

The direct base type of every enumeration type is the class System.Enum, while the direct base class of

System.Enum is System.ValueType.

##### Reference Types

A reference type is one of the following types:

* class type
* interface type
* array type
* delegate type

A reference type value is a reference to an instance of that type, known as an object. Null values are allowed for reference types and mean that there is no instance of the type.

reference-type: class-type interface-type array-type delegate-type

class-type:

type-name object string

interface-type: type-name

array-type:

non-array-type rank-specifiers

non-array-type: value-type class-type interface-type delegate-type type-parameter

rank-specifiers: rank-specifier

rank-specifiers rank-specifier

rank-specifier:

[ dim-separatorsopt ]

dim-separators:

,

dim-separators ,

delegate-type: type-name

Class Types

A class type is a data structure that contains the following:

* + **Data members.** These include constants and fields.
  + **Function members.** These include events, methods, properties, instance constructors, indexers, operators, finalizers, and static constructors.
  + Nested types.

*Note that class types do support inheritance.*

Object Type

The object class type is, ultimately, the base class of all other types and, every other type directly or indirectly derives from the object class type.

The object keyword is an alias for the System.Object class.

String Type

The string type is a sealed class that inherits directly from object. Instances of the string class repre- sent Unicode character strings and values of the string type can be written as string literals.

The string keyword is an alias for the System.String class.

Array Types

An array is a data structure. An array can contain zero or more variables that are accessed through indices. The variables contained in an array (also called the elements) must all be of the same type, called the element type of the array.

Delegate Types

A delegate is a data structure that refers to one or more methods. For instance, a delegate also refers to the corresponding object instances.

The null Type

The null literal evaluates to the null value, which is used to indicate a reference that doesn’t point to an object or array. It can also indicate the absence of a value.

The null type has a single value, which is the null value. This means that any expression that has a

null type can evaluate only to the null value.

Boxing and Unboxing

Boxing and unboxing are key components of C# types. They act as a pathway between value and refer- ence types by allowing value types to be converted to and from type object.

Boxing

A boxing conversion allows the programmer to implicitly convert any value type to object or System.ValueType or to any interface type implemented by the value type. There also exists an implicit boxing conversion from any enumeration type to System.Enum.

Boxing a value of a value type consists of allocating an object instance and copying the value type value into that instance.

Unboxing

An unboxing conversion allows the programmer to carry out an explicit conversion from object or System.ValueType to any value type, or from any interface type to any value type that implements the interface type. There is an explicit unboxing conversion from System.Enum to any enumeration type.

An unboxing operation consists of checking that the object instance is a boxed value of the given value type and then copying (not referring to) the value out of the instance.

Nullable Types

A nullable type is classed as a value type.

The type specified before the ? modifier in a nullable type is called the underlying type of the nullable type.

The underlying type of a nullable type can be any non-nullable value type or any type parameter limited to non-nullable value types.

The underlying type of a nullable type shall not be a nullable type or a reference type.

Members

An instance of a nullable type T? has two public properties that are read-only. These are:

* HasValue — The type of this property is bool.
* Value — The property is of type T.

For any instance where HasValue is true, it is said to be non-null. This instance will contain a value that will be returned by Value.

If HasValue is false, the instance is said to be null. Trying to read Value will cause a

System.InvalidOperationException to be thrown.

Every nullable type T? has a public constructor. This takes a single argument of type T. Given a value x of type T, the constructor invocation below creates a non-null instance of T? where the Value property is x.

new T? (x)

Implemented Interfaces

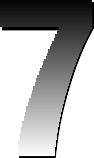
A type of the form T? implements the same interfaces as System.Nullable<T>.

This normally means that the interfaces implemented by T and T? are going to be different.

##### Summary

In this chapter you looked at a theme that is key to C# programming — types. This chapter has revolved around the fundamental difference between value types (where each variable has an independent copy of the data) and reference types (which refer to the same data).

In Chapter 7, you look at variables.



### Variables

In this chapter you look at a subject that is core to handling data of any kind in programming — variables. Variables are the cornerstone of handling and passing data in C# and other programming languages. Whenever there’s any data being handled or processed, variables are never far away!

##### What are Variables?

There are a number of ways to describe what a variable is. In cold computer terms, a variable is a storage location for data. Rather than having to mess around addressing memory locations directly (something that can lead even the best programmer into the tar pits), variables are referenced by the name attached to them.

You can think of variables as storage boxes in memory. Each box is given a name and can hold specific kinds of data, called values.

Every variable has a type. This type determines what values can be stored in the variable. C# is a type-safe language, and the compiler ensures that values stored in a variable are of the right type.

Not all Variables Are Created Equally

Not all variables are created in the same way. In fact, two kinds of variables can be created:

* + **Initially assigned.** Here are a few simple examples:

int myInt = 3;

string myString = “Hello”; char myChar = “x”;

* **Initially unassigned.** Here are a few simple examples:

int myInt; string myString; char myChar;

The difference between an initially assigned and an initially unassigned variable is that when an initially unassigned variable is created, it is created without an initial value, whereas an initially assigned vari- able has a well-defined initial value.

*A value has to be assigned to a variable before a value can be obtained from it (more on this later in this chapter).*

##### Categories of Variables

There are seven distinct categories of variables:

* Static variables
* Instance variables
* Array elements
* Value parameters
* Reference parameters
* Output parameters
* Local variables

All these variables will be discussed over the course of this chapter. All seven types of variables are shown in the following code snippet:

class VarEx

{

public static int StaticVar; int InstanceVar;

void F(int[] ArrayEl,

int ValueParam, ref int RefParam,

out int OutputParam) { int LocalVar = 1;

OutputVar = ValueParam + RefParam++;

}

}

Below is a list of the variable names used, along with the type of variable each name represents:

* + StaticVar — This is a static variable.
  + ArrayEl — This is an array element.
  + InstanceVar — This is an instance variable.
  + ValueParam — This is a value parameter.
  + RefParam — This is a reference parameter.
  + OutputParam — This is an output parameter.
  + LocalVar — This is a local variable.

Let’s take a look at each of these variable categories in turn.

Static Variables

Static variables are initially assigned variables.

Any field declared with a static modifier is called a static variable.

These variables come into being before the execution of a static constructor for the containing type. The variable disappears when the application domain it is associated with no longer exists.

The initial value of the static variable is the default value of the type of the variable.

class VarEx

{

public static int StaticVar; int InstanceVar;

void F(int[] ArrayEl,

int ValueParam, ref int RefParam,

out int OutputParam) { int LocalVar = 1;

OutputVar = ValueParam + RefParam++;

}

}

Array Elements

Array elements are initially assigned.

The elements of an array appear when the array instance is created and disappears when there is no longer any reference to that array instance.

The initial value of each array element is the default value of the type of the element.

class VarEx

{

public static int StaticVar; int InstanceVar;

void F(int[] ArrayEl, int ValueParam,

ref int RefParam,

out int OutputParam) { int LocalVar = 1;

OutputVar = ValueParam + RefParam++;

}

}

Instance Variables

Any field declared without the static modifier is known as an instance variable. Instance variables can be used in the following:

* Classes
* Structs

class VarEx

{

public static int StaticVar;

int InstanceVar;

void F(int[] ArrayEl,

int ValueParam, ref int RefParam,

out int OutputParam) { int LocalVar = 1;

OutputVar = ValueParam + RefParam++;

}

}

Using Instance Variables in Classes

Instance variables used in classes are initially assigned variables.

An instance variable of a class comes into being when a new instance of that class is created. The vari- able disappears when there are no longer any references to that instance (and any finalizers executed).

The initial value of any instance variable of a class is the default value of the variable type.

Using Instance Variables in Structs

Instance variables used in structs are initially assigned variables if the struct variable is assigned and are unassigned if the struct variable is unassigned.

Instance variables of structs have the same lifecycle as that of the struct itself. That is, they are created when the struct is created and disappear when the struct ends.

Value Parameter

Value parameters are initially assigned.

A value parameter is declared without a ref or out modifier.

The lifecycle of a value parameter starts when the function member (instance constructor, accessor, method, or operator) to which the parameter belongs is invoked. Value parameters are initialized with the value of the argument given during invocation.

Value parameters end on return of the function member (except where the parameter is captured by an anonymous method or the function member body is an iterator block).

class VarEx

{

public static int StaticVar; int InstanceVar;

void F(int[] ArrayEl,

int ValueParam, ref int RefParam,

out int OutputParam) { int LocalVar = 1;

OutputVar = ValueParam + RefParam++;

}

}

Reference Parameters

When within function members, reference parameters are initially assigned.

A parameter that has been declared with a ref modifier is called a reference parameter.

It is important to note that reference parameters don’t themselves create new storage locations in mem- ory. Instead, they are a representation of an existing storage location. This means that the value of a ref- erence parameter is always the same as that of the underlying variable.

class VarEx

{

public static int StaticVar; int InstanceVar;

void F(int[] ArrayEl,

int ValueParam,

ref int RefParam, out int OutputParam) {

int LocalVar = 1;

OutputVar = ValueParam + RefParam++;

}

}

Output Parameters

A parameter declared with an out modifier is called an output parameter.

As with reference parameters, output parameters do not create any new storage locations on memory. Output parameters reference the same storage location as the variable given as the argument in the func- tion member invocation.

Definite assignment rules are applicable to output parameters:

* No variable needs to be definitely assigned before it can be passed as an output parameter in a member invocation function.
* Within a function member, output parameters are initially unassigned.
* Output parameters of a function member have to be definitely assigned before the function member returns normally.

class VarEx

{

public static int StaticVar; int InstanceVar;

void F(int[] ArrayEl,

int ValueParam, ref int RefParam,

out int OutputParam) { int LocalVar = 1;

OutputVar = ValueParam + RefParam++;

}

}

Local Variables

Local variables are declared by:

* local-variable-declaration — The variable will be initially assigned.
* foreach-statement — Here the local variable is an exception variable.
  + specific-catch-clause of a try-statement — The variable will be initially assigned.

class VarEx

{

public static int StaticVar; int InstanceVar;

void F(int[] ArrayEl,

int ValueParam, ref int RefParam,

out int OutputParam) {

int LocalVar = 1; OutputVar = ValueParam + RefParam++;

}

}

##### Default Values

Variables belonging to the following categories are initialized to their default values automatically:

* + Static variables
  + Instance variables (of class instances)
  + Array elements

The default value of a variable depends on the type of the variable:

* + For a variable of a value-type, the default value will be the same as the value computed by the

value-type’s default constructor.

* + For reference-type, the default value is null.

##### Definite Assignment

If the compiler can prove that a variable has been automatically initialized or has been the target of one or more assignment, that variable is said to be definitely assigned.

There are a handful of rules for definite assignment:

* + Initially assigned variables are always considered to be definitely assigned.
  + Initially unassigned variables are considered to be definitely assigned if all the execution paths contain one of the following:
    - An invocation expression that passes the variable as an output parameter
    - An object-creation expression that passes the variable as an output parameter
* A simple assignment where the variable is a left operand
* A local variable declaration that includes a variable initializer (local variables only) Separate rules apply to struct-type variables and their instance variables:
* An instance variable is definitely assigned if the containing struct-type variable is definitely assigned.
* A struct-type variable is definitely assigned if each of the instance variables is also definitely assigned.

Initially Assigned Variables

The following variable categories are classified as initially assigned:

* Static variables
* Array elements
* Value parameters
* Reference parameters
* Instance variables of class instances
* Instance variables of initially assigned struct variables
* Variables declared by:
  + A using statement
  + A foreach statement
  + A catch clause

Initially Unassigned Variables

The following variable categories are initially unassigned:

* Instance variables of initially unassigned struct variables
* Local variables (except those declared in a foreach statement, a catch clause, or a using

statement)

* Output parameters

Rules for Determining Definite Assignment

The compiler uses specific rules to check whether a variable is definitely assigned or not.

To check, the compiler processes the body of each function that contains one (or more) unassigned vari- ables. For each such variable (v) encountered, the compiler defines the assignment state for the variable at the following spots:

* At the beginning of every statement
* At the end of every statement
  + A the point where control is transferred to another statement
  + At the beginning of every expressions
  + At the end of every expression

What follows are rules that control how the state of a variable is determined.

***General Rules for Statements***

* + v is not definitely assigned at the start of a function member body.
  + v is definitely assigned at the start of an unreachable statement.
  + The definite assignment state of v at the start of any other statement can be determined by checking the definite assignment state of v on all control-flow transfers that target the beginning of that statement.
  + The definite assignment state of v at the end of a block (checked, unchecked, if, while, do, for, foreach, lock, using, or switch statement) is determined by the compiler by checking the definite assignment state of v on all control-flow transfers that target the end of that statement.

***Rules for Block Statements, Checked, and Unchecked Statements***

* + The definite assignment state of v on the control transfer to the first statement of the statement list in the block will be the same as the definite assignment statement of v before the block, checked, or unchecked statement.

***Rules for Expression Statements***

The following rules apply for an expression statement stmt that consists of the expression expr:

* + v has the same assignment state at the beginning of expr as it does at the beginning of stmt.
  + When v is definitely assigned at the end of expr, it is definitely assigned at the end point of stmt.

***Rules for Declaration Statements***

* + If stmt is a declaration statement that does not have initializers, v will have the same definite assignment state at the end point of stmt as at the beginning of stmt.
  + If stmt is a declaration statement that does have initializers, the definite assignment state for v is determined as if stmt were a statement list, with one assignment statement for each declara- tion with an initializer.

***Rules for If Statements***

Let’s take a look at an if statement called stmt with the following form:

if ( expr ) then-stmt else else-stmt

* + v has the same definite assignment state at the beginning of expr as at the beginning of stmt.
  + If v is definitely assigned at the end of expr, it is also definitely assigned during the control-flow transfer to then-stmt and to either else-stmt or to the end of stmt if there is no else clause.
* If v is definitely assigned after an expression that returns a true at the end of expr, it is defi- nitely assigned during the control-flow transfer to then-stmt and not definitely assigned on the control-flow transfer to either else-stmt or to the end of stmt if there is no else clause.
* If v is definitely assigned after an expression that returns a false at the end of expr, it is defi- nitely assigned on the control-flow transfer to else-stmt and not definitely assigned on the control-flow transfer to then-stmt. It is definitely assigned at the end of stmt if and only if it is definitely assigned at the end-point of then-stmt.
* If none of the rules apply, v is not definitely assigned on the control-flow transfer to either the

then-stmt or else-stmt or to the end of stmt in the event that there is no else clause.

***Rules for Switch Statements***

In a switch statement, stmt that has the controlling expression expr:

* The definite assignment state of v at the beginning of expr is the same as the state of v at the beginning of stmt.
* The definite assignment state of v at control flow transfer to a switch block statement list is the same as the definite assignment state of v at the end of expr.

***Rules for While Statements***

Let’s take a while statement stmt of the form:

while ( expr ) while-body

* v has the same definite assignment state at the beginning of expr as it does at the beginning of

stmt.

* If v is definitely assigned at the end of expr, it is definitely assigned on the control-flow transfer to while-body and until the end of stmt.
* If v is definitely assigned after an expression that returns a true at the end of expr, it is defi- nitely assigned at the point of control-flow transfer to while-body but not definitely assigned at the end of stmt.
* If v is definitely assigned after an expression that returns a false at the end of expr, it is also def- initely assigned at the point of control-flow transfer to the end point of stmt but not definitely assigned on the control-flow transfer to while-body.

***Rules for Do Statements***

Let’s take a do statement stmt of the form:

do do-body while ( expr ) ;

* v has the same definite assignment state on the control-flow transfer from the beginning of stmt

to do-body as at the beginning of stmt.

* v has the same definite assignment state at the beginning of expr as it does at the end of do-body.
* If v is definitely assigned at the end of expr, it is definitely assigned on control-flow transfer to the end point of stmt.
  + If v is definitely assigned after an expression that returns a false at the end of expr, it is also definitely assigned on the control-flow transfer to the end point of stmt but is not definitely assigned on the control-flow transfer to do-body.

***Rules for Break, Continue, and Goto Statements***

* + The definite assignment state of v on the control-flow transfer caused by a break, continue, or

goto statement is the same as the definite assignment state of v at the beginning of the statement.

***Rules for Throw Statements***

Take a statement stmt of the form:

throw expr ;

* + The definite assignment state of v at the beginning of expr is the same as the definite assign- ment state of v at the beginning of stmt.

***Rules for Return Statements***

The rules for return statements depend on the form that the statement takes: For a statement stmt of the form:

return expr ;

* + The definite assignment state of v at the beginning of expr is the same as the definite assignment state of v at the beginning of stmt.
  + If v is an output parameter, it will be definitely assigned either:
    - After expr
    - At the end of the finally block of a try-finally or try-catch-finally that encloses the

return statement

If the statement stmt has the following form:

return ;

* + If v is an output parameter, it will be definitely assigned either:
    - Before stmt
    - At the end of the finally block of a try-finally or try-catch-finally that encloses the return statement

***Rules for Try-Catch Statements***

For a try-catch statement stmt of the form:

try try-block

catch ( ... ) catch-block-1

...

catch ( ... ) catch-block-n 93

* The definite assignment state of v at the beginning of try-block will be the same as the definite assignment state of v at the beginning of stmt.
* The definite assignment state of v at the beginning of catch-block-i is the same as the defi- nite assignment state of v at the beginning of stmt.
* The definite assignment state of v at the end-point of stmt is definitely assigned if v is definitely assigned at the end of try-block and every catch-block-i.

***Rules for Try-Finally Statements***

Let’s examine a try statement stmt of the form:

try try-block finally finally-block

* The definite assignment state of v at the beginning of try-block is the same as the definite assignment state of v at the beginning of stmt.
* The definite assignment state of v at the beginning of finally-block is the same as the defi- nite assignment state of v at the beginning of stmt.
* The definite assignment state of v at the end of stmt is definitely assigned if either:
  + v is definitely assigned at the end-point of try-block.
  + v is definitely assigned at the end-point of finally-block.

***Rules for Foreach Statements***

Let’s look at a foreach statement stmt of the form:

foreach ( type identifier in expr ) embedded-statement

* The definite assignment state of v at the beginning of expr is the same as the state of v at the beginning of stmt.
* The definite assignment state of v on the control-flow transfer to embedded-statement or to the end point of stmt will be the same as the state of v at the end of expr.

***Rules for Using Statements***

Let’s next take a look at a using statement stmt of the form:

using ( resource-acquisition ) embedded-statement

* The definite assignment state of v at the beginning of resource-acquisition is the same as the state of v at the beginning of stmt.
* The definite assignment state of v during the control-flow transfer to embedded-statement is the same as the state of v at the end of resource-acquisition.

***Rules for Lock Statements***

Next, a lock statement stmt of the form:

lock ( expr ) embedded-statement

* + The definite assignment state of v at the beginning of expr will be the same as the state of v at the beginning of stmt.
  + The definite assignment state of v during the control-flow transfer to embedded-statement is the same as the state of v at the end of expr.

***Rules for Simple Expressions***

The rules regarding simple expressions apply to the following expressions:

* + Literals
  + Simple names
  + Member access expressions
  + Nonindexed base access expressions
  + Typeof expressions

The definite assignment state of v at the end of the expression is the same as the definite assignment state of v at the beginning of the expression

The following rules:

* + The definite assignment state of v at the beginning of expr1 is the same as the definite assign- ment state at the beginning of expr.
  + The definite assignment state of v at the beginning of expri (where i is greater than one) is the same as the definite assignment state at the end of expri-1.
  + The definite assignment state of v at the end of expr is the same as the definite assignment state at the end of exprn.

Apply to these expressions:

* + Parenthesized expressions
  + Element access expressions
  + Base access expressions (with indexing)
  + Increment expressions
  + Decrement expressions
  + Cast expressions
  + unary +
  + -
  + ~
  + \* expressions
  + binary +
  + -
  + \*
* /
* %
* <<
* >>
* <
* <=
* >
* >=

❑ ==

* !=
* is
* as
* &
* |
* ^ expressions
* Compound assignment expressions
* Checked expressions
* Unchecked expressions
* Array
* Delegate creation expressions

***Rules for && Expressions***

Next, we’ll look at an expression expr of the form:

expr-first && expr-second

* The definite assignment state of v before expr-first will be the same as the definite assign- ment state of v before expr.
* The definite assignment state of v before expr-second will be definitely assigned if the state of v after expr-first is either definitely assigned or definitely assigned after a true expression. Otherwise, it will not be definitely assigned.
* The definite assignment state of v after expr is determined by:
  + If the state of v after expr-first is definitely assigned, the state of v after expr is also definitely assigned.
  + Otherwise, if the state of v after expr-second is definitely assigned and the state of v after expr-first is definitely assigned after false expression, the state of v after expr is defi- nitely assigned.
  + Otherwise, if the state of v after expr-second is definitely assigned or definitely assigned after a true expression, the state of v after expr is definitely assigned after true expression.
    - Otherwise, if the state of v after expr-first is definitely assigned after false expression and the state of v after expr-second is definitely assigned after false expression, the state of v after expr is definitely assigned after a false expression.
    - Otherwise, the state of v after expr is not definitely assigned.

***Rules for || Expressions***

Next, we’ll look at an expression expr of the form:

expr-first || expr-second

* The definite assignment state of v before expr-first will be the same as the definite assign- ment state of v before expr.
* The definite assignment state of v before expr-second will be definitely assigned if the state of v after expr-first is either definitely assigned or definitely assigned after a false expression. Otherwise, it will not be definitely assigned.
* The definite assignment state of v after expr is determined by:
  + If the state of v after expr-first is definitely assigned, the state of v after expr is also definitely assigned.
  + Otherwise, if the state of v after expr-second is definitely assigned and the state of v after expr-first is definitely assigned after a false expression, the state of v after expr is defi- nitely assigned.
  + Otherwise, if the state of v after expr-second is definitely assigned or definitely assigned after true expression, the state of v after expr is definitely assigned after a false expression.
  + Otherwise, if the state of v after expr-first is definitely assigned after a true expression and the state of v after expr-second is definitely assigned after a true expression, the state of v after expr is definitely assigned after a false expression.
  + Otherwise, the state of v after expr is not definitely assigned.

***Rules for ! Expressions***

For an expression expr of the form:

! expr-operand

* The definite assignment state of v before expr-operand is identical to the definite assignment state of v before expr.
* The definite assignment state of v after expr is determined by:
  + If the state of v after expr-operand is definitely assigned, the state of v after expr is definitely assigned.
  + If the state of v after expr-operand is not definitely assigned, the state of v after expr is also not definitely assigned.
  + If the state of v after expr-operand is definitely assigned after a false expression, the state of v after expr is definitely assigned after a true expression.
  + If the state of v after expr-operand is definitely assigned after a true expression, the state of v after expr is definitely assigned after a false expression.

***Rules for ?: Expressions***

For an expression expr of the form:

expr-cond ? expr-true : expr-false

* The definite assignment state of v before expr-cond will be the same as the state of v before expr.
* The definite assignment state of v before expr-true is definitely assigned if the state of v after

expr-cond is definitely assigned or definitely assigned after a true expression.

* The definite assignment state of v before expr-false is definitely assigned if the state of v after

expr-cond is definitely assigned or definitely assigned after a false expression.

* The definite assignment state of v after expr is determined by:
  + If expr-cond is a constant expression with a value true, the state of v after expr is the same as the state of v after expr-true.
  + Otherwise, if expr-cond is a constant expression with a value false, the state of v after

expr is the same as the state of v after expr-false.

* + Otherwise, if the state of v after expr-true is definitely assigned and the state of v after

expr-false is definitely assigned, the state of v after expr is definitely assigned.

* + Otherwise, the state of v after expr is not definitely assigned.

***Rules for Yield Statements***

Finally, let’s take a look at a yield return statement stmt of the form:

yield return expr ;

* A variable v has the same definite assignment state at the beginning of expr as at the beginning of stmt.
* If a variable v is definitely assigned at the end of expr, it is definitely assigned at the end of

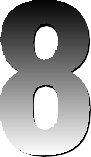
stmt. Otherwise, it is not definitely assigned at the end of stmt.

##### Summary

In this chapter you looked at one of the most important elements related to programming — variables. You learned about assigned and unassigned variables, along with the seven categories of variables.

After that you examined default values and definite assignment before looking in detail at the rules for definite assignment.

In Chapter 8, you look at conversions in C#.



### Conversions

In this chapter you look at conversions in C# and how they allow for flexibility when using types.

Conversions do one thing and one thing alone — allow an expression of one type to be treated as another type. Conversions can take one of two forms:

* + **Implicit.** These are conversions that can occur automatically as required within the code.
  + **Explicit.** These conversions require a cast to be called.

All conversions in C# must be static and must either take the type that the conversion is defined on or return that type.

int x = 01234;

long y = x; // this is an implicit conversion, from int to long

int z = (int) y; // this is an explicit conversion, from long to int

In the preceding example, there is a conversion from int to long. This is an implicit conversion, and expressions of the type int can be treated as though they have the type long. However, the reverse, a conversion from long to int, is an explicit conversion, and an explicit cast is needed for this to work.

##### Implicit Conversions

The following conversions are all considered implicit:

* + Identity conversions
  + Implicit numeric conversions
  + Implicit enumeration conversions
  + Implicit reference conversions
  + Boxing conversions
* Implicit type parameter conversions
* Implicit constant expression conversions
* User-defined implicit conversions

There are many situations where an implicit conversion can occur. For example, in:

* Assignments
* Function member invocations
* Cast expressions

Identity Conversions

An identity conversion involves a conversion from one type to the same type. Very little is useful about this. It serves as nothing more than a way of making sure that errors aren’t generated when trying to convert one type to the same type.

Implicit Numeric Conversions

The following are implicit numeric conversions:

* From sbyte to decimal, double, float, int, long, and short
* From long to double, decimal, or float
* From ulong to double, decimal, or float
* From char to double, decimal, float, ushort, int, uint, long, or ulong
* From float to double
* From byte to decimal, double, short, ushort, int, uint, long, ulong, or float
* From short to double, decimal, int, long, or float
* From ushort to double, decimal, int, uint, long, ulong, or float
* From int to double, decimal, long, or float
* From uint to double, decimal, long, ulong, or float

Conversions from int, uint, long or ulong to float and from long or ulong to double quite often cause a loss of precision in the resulting value. This should be borne in mind if you’re carrying out high- precision technical work. However, such conversions will never cause a loss of magnitude of the value (a number that has a magnitude that is 103 will still retain the same magnitude).

*No other implicit numeric conversions cause any loss of precision in the resulting value.*

It’s important to bear in mind that no implicit conversion to the char type is possible, and other integral values won’t automatically convert to this type (if you think about it, it wouldn’t make sense if they did, since character strings would make no sense as any other type).

Implicit Enumeration Conversions

Implicit enumeration conversions simply allow the decimal integer literal 0 to be converted to any enum

type without causing an error. The enum types are:

* + byte
  + sbyte
  + short
  + ushort
  + int
  + uint
  + long
  + ulong

Implicit Reference Conversions

The following are implicit reference conversions:

* + From any reference type to object
  + From any class type S to any class type T, provided S is derived from T
  + From any class type S to any interface type T, provided S implements T
  + From any interface type S to any interface type T, provided S is derived from T
  + From any array type to System.Array
  + From any delegate type to System.Delegate
  + From any array type to any interface implemented by System.Array
  + From any delegate type to System.ICloneable
  + From the null type to any reference type
  + From an array type S with an element type SE to an array type T with an element type TE, pro- vided all of the following are true:
    - S and T differ only in element type.
    - An implicit reference conversion exists from SE to TE.
  + From a one-dimensional array type S[] to System.Collections.Generic.IList<S> and base interfaces of this interface
  + From a one-dimensional array type S[] to System.Collections.Generic.IList<T> and base interfaces of this interface (if there is an implicit reference conversion from S to T)

If the type parameter is known to be a reference type, the following implicit references exist:

* From the null type to T
* From T to its effective base class C, from T to any base class of C, and from T to any interface implemented by C
* From T to an interface type I in T’s effective interface set and from T to any base interface of I
* From T to a type parameter U, provided that T depends on U

Boxing Conversions

A boxing conversion allows any value type to be implicitly converted as follows:

* To the type object
* To System.ValueType
* To any interface type implemented by the value type

It also allows any enum type to be implicitly converted to System.Enum. Boxing a value of a value type consists of:

* Allocating an object instance
* Copying the value type value into that instance A few additional notes:
* An enum can be boxed to the type System.Enum, because it is the direct base class for all enums.
* A struct or enum can be boxed to the type System.ValueType, because that is the direct base class for all structs and a base class for all enums.

For any type parameter T that is not a reference type, the following are all considered to be boxing con- versions:

* From T to its effective base class C, from T to any base class C, and from T to any interface imple- mented by C
* From T to an interface type I in T’s interface set and from T to any base interface of I

Implicit Type Parameter Conversions

For a type parameter T that is not known to be a reference type, there will be an implicit conversion from

T to a type parameter U, provided that the type parameter T depends on U.

At runtime, if T is a value type and U is a reference type, the conversion will be carried out as though it is a boxing conversion.

At runtime, if both T and U are value types, T and U are necessarily the same type, and no conversion will be carried out on either of the types.

At runtime, if T is a reference type, U will also be a reference type, and the conversion is carried out as either an implicit reference conversion or an identity conversion.

Implicit Constant Expression Conversions

An implicit conversion expression allows for the following conversions to be carried out:

* + Any constant expression of the type int can be converted to byte, sbyte, short, ushort, uint, or ulong as long as the value of the constant expression is within the range of the result- ing type.
  + Any constant expression of the type long can be converted to the type ulong, as long as the value of the constant expression is not negative.

User Defined Implicit Conversions

A user-defined implicit conversion consists of:

* + An optional standard implicit conversion, followed by
  + The execution of a user-defined implicit conversion operator, followed by
  + Another optional standard implicit conversion

##### Explicit Conversions

Explicit conversions are classed as follows:

* + All implicit conversions
  + Explicit numeric conversions
  + Explicit enumeration conversions
  + Explicit reference conversions
  + Unboxing conversions
  + Explicit type parameter conversions
  + User-defined explicit conversions

Explicit Numeric Conversions

Explicit numeric conversions are conversions from one numeric type to another where an implicit con- version does not exist:

* + From sbyte to byte, ushort, uint, ulong, or char
  + From byte to sbyte or char
* From short to sbyte, byte, ushort, uint, ulong, or char
* From ushort to sbyte, byte, short, or char
* From int to sbyte, byte, short, ushort, uint, ulong, or char
* From uint to sbyte, byte, short, ushort, int, or char
* From long to sbyte, byte, short, ushort, int, uint, ulong, or char
* From ulong to sbyte, byte, short, ushort, int, uint, long, or char
* From char to sbyte, byte, or short
* From float to sbyte, byte, short, ushort, int, uint, long, ulong, char, or decimal
* From double to sbyte, byte, short, ushort, int, uint, long, ulong, char, float, or decimal
* From decimal to sbyte, byte, short, ushort, int, uint, long, ulong, char, float, or double

*Because explicit conversions cover all implicit and explicit numerical conversions, it is always possible to convert from one numeric type to anther using a cast expression (covered in greater detail in Chapter 9).*

Using explicit numeric conversions can sometimes cause a loss of information; bear this in mind if high precision is important. It is also possible for explicit numeric conversions to throw an exception.

Explicit numeric conversions are processed depending on the type of conversion being carried out.

Integral Type to Integral Type

This conversion depends on the overflow-checking context in which the conversion takes place, which we will now look at.

* When carried out in a checked context, the conversion will be successful if the value of the source operand falls within the range of the destination type. A System.OverflowException is thrown if the value of the source operand falls outside the range of the destination type.
* When carried out in an unchecked context, the conversion will always be successful. The fol- lowing processes will be carried out:
  + If the source type is larger than the destination type, the source is truncated by discarding significant bits.
  + If the source is smaller, the source value is sign-extended if the source type is signed (sim- ply put, this means that the + or – is added) or zero-extended if it is unsigned.
  + If the source type is identical to the destination type, they are treated as equivalent.

Decimal to Integral Type

In conversions that go from decimal to an integral type, the source type is always rounded — toward zero — to the nearest integral value. This integer becomes the result of the conversion. There is signifi- cant loss of precision here.

If the resulting integral value falls outside of the range of the destination type, the conversion results in a

System.OverflowException being thrown.

Float/Double to Int Type

Conversion from float to int and double to int depends on the overflow-checking context in which the conversion takes place.

* + - In a checked context, the value is rounded — toward zero — to the nearest negative integral value. If this resulting integral value falls within the range of the destination type, the value is the result of the conversion. If it falls outside, a System.OverflowException is thrown.
    - In an unchecked context, the conversion will always be successful. The value is rounded — toward zero — to the nearest integral value. If this value falls within the range of the destination type, this becomes the value of the conversion; otherwise, the result of the conversion is an unspecified value.

Double to Float

In conversions from double to float, the double value is rounded to the nearest float value.

*Be aware that this rounding may cause a value that is initially nonzero to be rounded to a zero value.*

Double values that are too big to be represented as a float will result in a positive infinity or negative infinity value.

If the double value is NaN, the result of this conversion will also be NaN.

Float/Double to Decimal

In conversions from float or double to decimal, the source values will be converted to decimal and then subsequently rounded to the nearest number. This rounding might cause a nonzero number to be rounded to zero, which will result in a significant loss of precision.

If the source number is too large to be represented as decimal or if the value is either NaN or infinity, a

System.OverflowException will be thrown.

Decimal to Float/Double

In conversions that involve a conversion from decimal to float or double, the value is rounded to the nearest float or double value as required by the code.

If the value being converted does not fall within the range of the destination type, a

System.OverflowException is thrown.

Explicit Enumeration Conversions

Explicit enumeration conversions are:

* + - From sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, or decimal

to any enum type

* + - From any enum type to sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, or decimal
    - From any enum type to any other enum type

Explicit Reference Conversions

Explicit reference conversions are:

* From object to any reference type
* From any class type S to any class type T, as long as S is a base class of T
* From any class type S to any interface type T, as long as S is not sealed and provided S does not implement T
* From any interface type S to any class type T, as long as T is not sealed or provided T

implements S

* From any interface type S to any interface type T, as long as S is not derived from T
* From System.Array and the interfaces it implements, to any array type
* From System.Delegate and the interfaces it implements, to any delegate type
* From a one-dimensional array type S[] to System.Collections.Generic.IList<T> and its base interfaces, as long as there is an explicit reference conversion from S to T
* From System.Collections.Generic.IList<T> and its base interfaces to a one-dimensional array type S[], as long as there is an implicit or explicit reference conversion from S[] to System.Collections.Generic.IList<T>
* From an array type S with an element type SE to an array type T with an element type TE, as long as all of the following are true:
  + S and T differ only in element type.
  + An explicit reference conversion exists from SE to TE.

For a type-parameter T which is a reference type, the following explicit reference conversions are allowable:

* From the effective base class C of T to T and from any base class of C to T
* From any interface type to T
* From T to any interface type I, as long as there isn’t already an implicit reference conversion from T to I
* From a type parameter U to T, as long as T depends on U

Explicit reference conversions are carried out between reference types that require runtime checks to ensure they are correct.

For an explicit reference conversion to be successful during runtime, the value of the source operand must be null, or the runtime type of the object referenced by the source operand has to be a type that can be converted to the destination type by an implicit reference conversion.

If an explicit reference conversion is unsuccessful, a System.InvalidCastException is thrown.

Unboxing Conversions

An unboxing conversion allows:

* + An explicit conversion from type object to System.ValueType to a value type
  + From an interface type to any value type that implements the interface type
  + From the type System.Enum to any enumeration type

An unboxing operation is a two-step process and proceeds as follows:

* + A check is carried out to make sure that the object instance is a boxed value of a given value or enumeration type.
  + The value is copied from the instance.

Explicit Type Parameter Conversions

For a type parameter T that is not known to be a reference type, the following explicit conversions are allowed:

* + From T to any interface type I, provided there is not already an implicit conversion from T to I
  + From a type parameter U to T, provided that T depends on U

User-Defined Explicit Conversions

User-defined explicit conversions are made up of:

* + An optional explicit conversion, followed by
  + The execution of a user-defined implicit or explicit conversion operator, followed by
  + Another optional standard explicit conversion

##### Standard Conversions

The standard conversions, explained in the following sections, are predefined and can occur as part of a user-defined conversion.

Standard Implicit Conversions

The following conversions are all standard implicit conversions:

* + Identity conversions
  + Implicit numeric conversions
  + Implicit reference conversions
  + Boxing conversions
* Implicit type parameter conversions
* Implicit constant expression conversions
* Implicit nullable conversions

Standard Explicit Conversions

The standard explicit conversions are all standard implicit conversions, along with the subset of the explicit conversions for which an opposite standard implicit conversion exists.

User-Defined Conversions

C# allows for predefined implicit and explicit conversions to be augmented by user-defined conversions. This is carried out by declaring conversion operators in class and struct types.

*It is not possible to redefine a conversion already defined as implicit or explicit.*

User-Defined Implicit Conversions

User-defined implicit conversions from type S to type T are carried out as follows:

* Find the types S0 and T0 that result from deleting the trailing ? modifiers from S and T.
* Find the set of types, D, from which user-defined conversion operators will be considered. This set consists of S0, which is a class or struct, the base classes of S0 if S0 is a class, and T0 if T0 is a class or struct.
* Discover the set of applicable conversion operators, U. This is made up of the user-defined and, if S and T are both nullable, lifted implicit conversion operators declared by the classes or structs in D that convert from a type encompassing S to a type encompassed by T.

*If* U *is empty, there is no conversion, and a compile-time error occurs.*

* Work out the most specific source type, SX, of the operators in U. If any of the operators in U con- vert from S, SX is S; otherwise, SX is the most encompassed type in the combined set of source types of the operators in U.
* Work out the most specific target type, TX, from the operators in U. If any of the operators in U convert to T, TX is T; otherwise, TX is the most encompassed type in the combined set of target types of the operators in U.
* Work out the most specific conversion operator. If U contains exactly one user-defined conver- sion operator that converts SX to TX, this is the most specific; otherwise, if U contains one lifted conversion operator that converts from SX to TX, this is the most specific conversion operator. If the conversion is ambiguous, a compile-time error occurs.
  + Finally, the conversions are applied as follows:
  + If S is not SX, a standard implicit conversion from S to SX is carried out.
  + The most specific conversion is invoked and converted from SX to TX.
  + If TX is not T, a standard implicit conversion from TX to T is carried out.

##### Anonymous Method Conversions

An implicit conversion exists from an anonymous method expression to any compatible delegate type. If D is a delegate type and A is an anonymous method expression, D is compatible with A if the following conditions are true:

* + - If A does not contain an anonymous method signature, D can have zero or more parameters of any type as long as the out parameter of D is modified.
    - If A has an anonymous method signature, D will have the same number of parameters. Each parameter of A will be compatible with the corresponding parameter on D (this occurs when they are both of the same type and when the presence or absence of the out or ref modifiers on A match those of D).

##### Method Group Conversions

Method group conversions are implicit conversion methods that transform them to a compatible dele- gate. This is similar to the implicit anonymous group method.

If D is a delegate type and E is an expression classified as a method group, D will be compatible with E if (and only if) E contains at least one method that is applicable in its normal form to any argument list having types and modifiers matching the parameter types and modifiers of D.

The compile-time application of this conversion of E to D is identical to the compile-time processing of the delegate creating expression D(E).

##### Null Type Conversions

An implicit conversion is allowed from the null type to any nullable type. This conversion will produce a null value of the given nullable type.

Nullable Conversions

Before we look at nullable conversions, allow us to introduce two terms:

* + - **Wrapping.** This is a process of packaging a value of type T in an instance of type T?. A value x

of type T is wrapped to type T? by evaluating a new expression: T?(x).

* + - **Unwrapping.** This is the process of returning a value of type T contained in an instance of type T?. This is done by evaluating the expression x.Value. Unwrapping null instances will cause a System.InvalidOperationException to be thrown.

Nullable conversions allow for predefined conversions that work on non-nullable values types. Each predefined conversion converts from a nullable value type S to a non-nullable value T.

For every predefined implicit or explicit conversion that converts from a non-nullable value type S to a non-nullable value T, the following must exist:

* There must be either an implicit or explicit nullable conversion from S? to T?.
* There must be an implicit or explicit nullable conversion from S to T?.
* There must be an explicit nullable conversion from S? to T.

In the preceding, a nullable conversion can be either an implicit or explicit conversion.

##### Summary

This chapter looked in detail at both implicit and explicit conversions in C#. As a standalone chapter, the content here might seem complex, which is why it’s recommended that you read this chapter as part of a bigger reading plan and read the referenced chapters too.

In Chapter 9, you look at expressions in C#.



### Expressions

In this chapter you take a detailed look at expressions in C#. Expressions are at the core of all coding that you will do, so we will take quite some time to work through the different kinds of expressions allowed in C#.

Any valid sequence of operators and operands is called an expression. Expressions have a specific order for evaluating of the operands and operators. Also, different expressions will have different meanings.

##### Classifications of Expressions

There are a number of different classifications of expressions. Each expression falls into one category:

* + **Value.** Every value will have an associated type.
  + **Variable.** Every variable will have an associated type, the declared type of the variable.
  + **Namespace.** Expressions with the namespace classification can only appear on the left- hand side for a member access.
  + **Type.** Expressions with the type classification can only appear on the left-hand side for a member access.
  + **Method group.** These are overloaded methods that result from member lookup.
  + **Anonymous method.** These are expressions used on a delegate creation expression or implicitly converted to a compatible delegate type.
  + **Property access.** Every property access used has an associated type, which will be the type of the property.
  + **Event access.** Every event access used has an associated type, which will be the type of the event.
  + **Indexer access.** Every indexer access used has an associated type, which will be the ele- ment type of the indexer.

*When an expression is an invocation of any method with a return type of* void*, the expression is classi- fied as having no classification at all — a nothing.*

Results of an Expression

The result of an expression cannot be any of the following:

* Anonymous method
* Event access
* Method group
* Namespace
* Type

Instead, these categories are merely intermediates used in specific contexts.

##### Expression Values

Most expressions invariably end up with a value. Since this is the case, if the expression denotes a namespace, a nothing, or a type, a compiler error is generated.

If an expression denotes a variable, indexer, or property access, the value will be implicitly and automat- ically substituted. Let’s take a look at the rules that encompass this:

* **Variables.** Values of variables will be the value stored in the variable.
* **Indexers.** This value is obtained by invoking the get-accessor of the indexer. If no get- accessor exists, a compiler-time error results.
* **Property access.** This value is obtained by invoking the get-accessor of the property access. If no get-accessor exists, a compiler-time error results.

##### Expressions and Operators

All expressions are made up of operands and operators. Operands are the inputs to the operators, and the operators are used to indicate what operations should be applied to the operands. The following table provides an example.

|  |  |  |
| --- | --- | --- |
| **operand** | **operator** | **operand** |
| 2 | + | 5 |

The commonest types of operators are mathematical operators such as +, -, \*, and /. The commonest types of operands in C# include variables, constants, and expressions.

Three Kinds of Operator

There are three kinds of operators:

* + **Unary.** Takes one operand and uses either prefix (-x) or postfix (x++) notation
  + **Binary.** Takes two operands and all use infix notation (that is, they go between the operands)

x + y y - z

* + **Ternary.** There is only one ternary operator, ?:. This takes three operands, and it uses infix notation.

(z ? x : y)

This is a handy shorthand way of saying:

condition ? value if true : value if false

In expressions, the order of evaluation is controlled by both the precedence and associativity of the oper- ators (discussed in more detail in the following section).

Operands are processed left to right:

4 + 4 + 3

4 + 4 = 8 + 3 = 11

This order can be overridden using parentheses:

4 + (4 \* 3)

(4 \* 3) = 12 + 4 = 16

Note that this expression:

4 + 4 \* 3

Is the same as:

4 + (4 \* 3)

Operator Precedence and Associativity

Expressions that contain more than one operator rely on operator precedence to control the order in which the operators are evaluated.

Here is a table that lists the operator precedence for all operators, from high to low:

**Category Operators**

Primary

Unary

Multiplicative

Additive Shift

Relational and type-testing

Equality

Logical AND Logical XOR Logical OR Conditional AND Conditional OR Null Coalesing

x.y f(x)

a[x] x++ x-- new

typeof checked unchecked

+

-

!

~

++x (T)x

\*

/

%

+

-

<<

>>

<

>

<=

>=

is as

==

!=

&

^

| &&

||

??

**Category Operators**

Assignment ?:

\*=

/=

%=

+=

-=

<<=

>>=

&=

^=

|=

When operands are between two operators and these two operators have the same precedence value, associativity is used to control the order of processing.

These are the rules of associativity:

* + Apart from assignment and null coalescing operators, all other binary operators are left associa- tive. That means that operations are carried out left to right.
  + Assignment, null coalescing, and the single ternary operator (the conditional operator) are right associative. This means that operations are carried out right to left.

Operator Overloading

All unary and binary operators have a predefined set of implementations available by default (that is, the + operator can carry out addition, the – subtraction, and so on) in any expression they are used in. To augment these predefined implementations, user-defined implementations can be introduced by includ- ing operator declarations in classes and structs.

User-defined operator implementations always take precedence over predefined operator implementa- tions. Only when there is no applicable user-defined operator implementation are predefined operator implementations used.

Overloadable unary operators are:

❑ +

* + -
  + !
  + ~

❑ ++

* + --
  + true
  + false

Overloadable binary operators are:

❑ +

* -
* \*
* /
* %
* &
* |
* ^
* <<
* >>

❑ ==

* !=
* >
* <
* >=
* <=

It is important to note that when any binary operator is overloaded, the associated assignment operator, if it exists, is implicitly overloaded.

In expressions, operators are referenced using operator notation, but in declarations, operators are refer- enced using functional notation. The following table shows the relationship between operator and func- tional notations for unary and binary operators.

**Operator notation Functional notation**

op x

Here op denotes any overloadable unary prefix operator.

x op

Here op denotes the unary postfix ++ and – operators.

x op y

Here op denotes any overloadable binary operator.

operator op(x)

operator op(x)

operator op(x,y)

User-defined operator declarations require one or more of the parameters to be of the class or struct type that contains the operator declaration.

User-defined operator declarations cannot modify any of the following aspects of an operator:

* Associativity
* Precedence
* Syntax

Unary Operator Overload Resolution

An operation that takes on the form op x or x op (where op is an overloadable unary operator and x is an expression of type X) is processed using the follow rules:

* The set of candidate user-defined operators provided by X for the operation operator op(x) is determined using the following rules:

Given a type T and an operation operator op(A), where op is an overloadable operator and A is an argument list, the set of candidate user-defined operators provided by T for operator op(A) is determined as follows:

* + Determine the type T0 that results from removing the trailing ? modifiers, if any, from T.
  + For all operator op declarations in T0, if at least one operator is applicable with respect to the argument list A, the set of candidate operators consists of all applicable operator op declarations in T0. The lifted forms of the operators declared in T0 are considered also to be declared by T0.
  + Alternatively, if T0 is object, the set of candidate operators is empty.
  + Alternatively, the set of candidate operators provided by T0 is the set of candidate opera- tors provided by the direct base class of T0.
* If the set of candidate user-defined operators is not empty, these are then set as the candidate operators for the operation. Otherwise, the predefined unary operator op implementations become the candidate operators for the operation. If type X is not an enum type, any predefined unary operator with a parameter type that is an enum type is not considered.
* The following overload resolution rules are applied to the set of candidate operators to select the most appropriate operator with regard to the argument list (x). This operator becomes the result of the overload resolution process. Given a type T and an operation operator op(A), where op is an overloadable operator and A is an argument list, the set of candidate user- defined operators provided by T for operator op(A) is determined as follows:
  + Determine the type T0 that results from removing the trailing ? modifiers, if any, from T.
  + For all operator op declarations in T0, if at least one operator is applicable with respect to the argument list A, the set of candidate operators consists of all applicable operator op declarations in T0. The lifted forms of the operators declared in T0 are considered also to be declared by T0.
  + Alternatively, if T0 is object, the set of candidate operators is empty.
  + Alternatively, the set of candidate operators provided by T0 is the set of candidate opera- tors provided by the direct base class of T0.
* If overload resolution fails to select a best operator, a compiler error is generated.

Binary Operator Overload Resolution

An operation of the form x op y, where op is an overloadable binary operator, x is an expression of type X, and y is an expression of type Y, will be processed according to the following rules:

* The set of candidate user-defined operators provided by X and Y for the operation operator op(x, y) are determined. The set consists of the union of the candidate operators provided by X and the candidate operators provided by Y, each determined using the rules which follow:
  + Determine the type T0 that results from removing the trailing ? modifiers, if any, from T.
  + For all operator op declarations in T0, if at least one operator is applicable with respect to the argument list A, the set of candidate operators consists of all applicable operator op declarations in T0. The lifted forms of the operators declared in T0 are considered also to be declared by T0.
  + Alternatively, if T0 is an object, the set of candidate operators is empty.
  + Alternatively, the set of candidate operators provided by T0 is the set of candidate opera- tors provided by the direct base class of T0.
* If the set of candidate user-defined operators is not empty, this is set as the candidate operators for the operation. If it is empty, the predefined binary operator op implementations become the set of candidate operators for the operation.
* The overload resolution rules (listed above) are applied to the set of candidate operators to select the best operator with respect to the argument list (x, y), and this operator becomes the result of the overload resolution process.
* If overload resolution fails to select a best operator, a compiler error is generated.

Lifted Operators

Lifted operators allow predefined and user-defined operators that operate on non-nullable value types to be used with nullable forms of those types. Lifted operators are formed from predefined and user-defined operators. These operators, however, do have to meet certain requirements, discussed as follows.

Unary Operators

The unary operators are:

❑ +

❑ ++

* -
* --
* !
* ~

An operator exists in a lifted form if the operand and result types are both non-nullable value types. The lifted form is constructed by adding a single ? modifier to the operand and result types (for example, !?).

The lifted operator produces a null value when the operand is null.

Equality Operators

The equality operators are:

❑ ==

* + !=

For equality operators, a lifted form of an operator exists if the operand types are both non-nullable value types and if the result type is bool.

The lifted forms are created by adding a single ? modifier to each operand type.

Relational Operators

The relational operators are:

* + <
  + >
  + <=
  + >=

The lifted form of a relational operator exists if the operand types are both non-nullable value types and if the result type is bool.

The lifted form is constructed by adding a single ? modifier to each operand type. The lifted operator produces the value false if one or both operands are null.

##### Member Lookup

A member lookup happens when the meaning of a name in the context of the type must be determined. A member lookup can happen as part of evaluating a simple-name or a member-access in an expression.

Member lookup takes into account not only the name of a member but also the number of type parame- ters the member has. It also looks at whether the member is accessible. For the purposes of member lookup, both generic methods and nested generic types have the number of type parameters that are indicated in their respective declarations and all other members will have zero type parameters.

A member lookup of a name N with K type parameters in a type T is processed in the following way:

* + The set of accessible members named N is worked out:
    - If T is a type parameter, then the set is the union of the sets of accessible members named N in each of the types specified as a primary constraint or secondary constraint for T, com- bined with the set of accessible members named N in object.
    - Alternatively, the set consists of all accessible members named N in T (which includes inherited members and the accessible members named N in object). If T is a constructed type, the set of members is obtained by substituting type arguments. Members that include an override modifier are excluded from the set.
* If the set of accessible members is empty, the member lookup does not produce a match, and no further steps are made.
* If K is zero, all nested types whose declaration included type parameters are removed. If K is not zero, all members with a different number of type parameters are removed.
* The members hidden by other members are also removed from the set. For every member S.M in the set, where S is the type in which the member M is declared, the following set of rules is applied:
  + If M is a constant, enumeration member, event, field, property, or type declaration, all members declared in a base type of S will be removed from the set.
  + If M is a method, all nonmethod members declared in a base type of S are removed.
* The interface members hidden by class members are next removed from the set. For every mem- ber S.M in the set, where S is the type in which the member M is declared, the following rules are applied if S is a class declaration other than object:
  + If M is a constant, event, enumeration member, field, property, or type declaration, all members declared in the interface declaration will be removed from the set.
  + If M is a method, all nonmethod members declared in an interface declaration are removed.
* Finally, the result of the lookup is determined:
  + If the set is made up of a single member that is not a method, this member will become the result of the lookup.
  + If the set contains nothing but methods, the group of methods is the result of the lookup.
  + Otherwise, the lookup is ambiguous, and a compiler error is generated.

Base Types

For member lookups, a type T will have the following base types:

* If T is object, T has no base type.
* If T is an enum type, the base types of T are the class types System.Enum, System.ValueType, and object.
* If T is a struct type, the base types of T are the class types System.ValueType and object.
* If T is a class type, the base types of T are the base classes of T, including the class type object.
* If T is an interface type, the base types of T are the base interfaces of T and the class type object.
* If T is an array type, the base types of T are the class types System.Array and object.
* If T is a delegate type, the base types of T are the class types System.Delegate and object.
* If T is a nullable type, the base types of T are the class types System.ValueType and object.

##### Function Members

Function members contain executable statements, are always members of types, and cannot be members of namespaces.

C# defines the following categories of function members:

* + Methods
  + Properties
  + Events
  + Indexers
  + User-defined operators
  + Instance constructors
  + Static constructors
  + Finalizers

Following are tables that summarize the processing that takes place in constructs involving each of the six categories of function members that can be explicitly invoked.

*Note that* e*,* x*,* y*, and* value *indicate expressions classified as variables or values,* T *indicates an expression classified as a type,* F *is the simple name of a method, and* P *is the simple name of a property.*

**Example Description**

**Method Invocation**

The method is invoked with the instance expression e and the argument list (x, y).

F(x,y)

Overload resolution is used to select the best method F in the con- taining class or struct.

The method is invoked with the argument list (x, y).

If the method is not static, the instance expression is this.

T.F(x,y)

Overload resolution is used to select the best method F in the class or struct T.

A compiler error is generated if the method is not static.

e. F(x,y)

The method is invoked with the argument list (x, y).

Overload resolution is used to select the best method F in the class, struct, or interface given by the type of e.

A compiler error is generated if the method is static.

**Example Description**

**Property Access**

P The get accessor of the property P in the containing class or struct is invoked.

A compiler error is generated if P is write-only.

P=value

If P is not static, the instance expression is this.

The set accessor of the property P in the containing class or struct is invoked with the argument list (value).

A compiler error is generated if P is read-only.

If P is not static, the instance expression is this.

T.P The get accessor of the property P in the class or struct T is invoked.

T.P=value

A compiler error is generated if P is not static or if P is write-only.

The set accessor of the property P in the class or struct T is invoked with the argument list (value).

A compile-time error occurs if P is not static or if P is read-only.

e.P The get accessor of the property P in the class, struct, or interface given by the type of e is invoked with the instance expression e.

e.P=value

**Event Access**

A compiler error is generated if P is static or if P is write-only.

The set accessor of the property P in the class, struct, or interface given by the type of e is invoked with the instance expression e and the argument list (value).

A compiler error is generated if P is static or if P is read-only.

E +=value

The add accessor of the event E in the containing class or struct is invoked.

If E is not static, the instance expression is this.

E -= value

The remove accessor of the event E in the containing class or struct is invoked.

If E is not static, the instance expression is this.

**Example Description**

T.E+=value

The add accessor of the event E in the class or struct T is invoked.

T.E-=value

A compiler error is generated if E is not static.

The get accessor of the event E in the class or struct T is invoked.

e.E+=value

A compiler error is generated if E is not static.

The add accessor of the event E in the class, struct, or interface given by the type of e is invoked with the instance expression e.

e.E-=value

A compiler error is generated if E is static.

The remove accessor of the event E in the class, struct, or interface given by the type of e is invoked with the instance expression e.

**Indexer Access**

A compile-time error occurs if E is static.

e[x, y]

Overload resolution is used to select the most appropriate indexer in the class, struct, or interface given by the type of e.

The get accessor of the indexer is invoked with the instance expression e and the argument list (x, y).

e[x, y]=value

A compiler error is generated if the indexer is set to write-only.

Overload resolution is used to select the most appropriate indexer in the class, struct, or interface given by the type of e.

The set accessor of the indexer is invoked with the instance expression e and the argument list (x, y, value).

A compiler error is generated if the indexer is read-only.

**Operator Invocation**

-x

x+y

Overload resolution is used to select the best unary operator in the class or struct given by the type of x.

Overload resolution is used to select the best binary operator in the classes or structs given by the types of x and y.

**Instance Constructor Invocation**

New T(x,y) Overload resolution is used to select the most appropriate instance constructor in the class or struct T.

**Argument Lists**

Every function member invocation will include an argument list. This list provides the values or variable references used by the parameters of the function member.

The syntax used for specifying the argument list will depend on the function member category. The following are rules for determining the argument list:

* For all the following, arguments are specified as an argument list (detailed later):
  + Delegates
  + Instance constructors
  + Methods
* For all properties, the argument list is empty when invoking the get accessor.
* For events, the argument list will be made up of the expression that appears as the right operand of the += or -= operator.
* For all indexers, the argument list is made up of the expressions specified between the square brackets ([ and ]) in the indexer access.
* For any user-defined operators, the argument list will be made up of the single operand of the unary operator or the two operands of the binary operator.

The arguments of the following are always passed as value parameters:

* Events
* Properties
* User-defined operators

Arguments of indexers are passed as value parameters or parameter arrays. Here is the structure of an argument list:

argument-list: argument

argument-list , argument

argument:

expression

ref variable-reference out variable-reference

An argument list is made up of one or more arguments. These arguments are separated by commas. Each argument can take one of the following forms:

124

* An expression used to indicate that the argument is passed as a value parameter
* The keyword ref followed by a variable-reference, which indicates that the argument is passed as a reference parameter
* The keyword out followed by a variable-reference, used to indicate that the argument is passed as an output parameter

Overload Resolution

Overload resolution is a mechanism used by the C# compiler that allows it to select the most appropriate function member to invoke given an argument list and a set of candidate function members.

Overload resolution selects the function member to invoke in the following way:

* + Invocation of a method named in an invocation expression
  + Invocation of an instance constructor named in an object-creation expression
  + Invocation of an indexer accessor through an element access
  + Invocation of a predefined or user-defined operator referenced in an expression

##### Primary Expressions

Primary expressions are made up of the simplest types of expression that can be found in C#:

primary-expression:

array-creation-expression

primary-no-array-creation-expression

primary-no-array-creation-expression: literal

simple-name parenthesized-expression member-access

invocation-expression element-access

this-access base-access

post-increment-expression post-decrement-expression object-creation-expression delegate-creation-expression typeof-expression

checked-expression unchecked-expression default-value-expression anonymous-method-expression

Literals

A primary expression made up of a literal will be classified as a value:

literal::

boolean-literal integer-literal real-literal character-literal string-literal null-literal

Simple Names

A simple name is made up of an identifier.

This identifier can be followed by a type argument list:

simple-name:

identifier type-argument-listopt

Parenthesized Expressions

A parenthesized expression is simply enclosed by parentheses:

parenthesized-expression: ( expression )

There’s very little to a parenthesized expression — the expression inside the parentheses is evaluated. The expression cannot denote a namespace or a type; otherwise, an error will be generated.

Member Access

A member access consists of either:

* A primary expression
* A predefined type
* Or a qualified-alias-member These will be followed by
* A “.” token
* An identifier
* And finally, optionally followed by a type argument list The following shows the syntax of the code that will be used:

member-access:

primary-expression . identifier type-argument-listopt predefined-type . identifier type-argument-listopt qualified-alias-member . identifier type-argument-listopt

predefined-type: one of bool

byte char decimal double float int

long object sbyte short string uint ulong ushort

A member access can take on either of the following forms:

* + E.I
  + E.I<A1, ..., AK>

E is a primary expression, predefined type, or qualified-alias-member; I is a single identifier, and <A1,

..., AK> is an optional type argument list.

Invocation Expressions

Invocation lists are used to invoke methods:

invocation-expression:

primary-expression ( argument-listopt )

The primary expression of an invocation expression is either a method group or a value of a delegate type.

If the primary expression is a method group, the invocation expression is a method invocation. If the pri- mary expression is a value of a delegate type, the invocation expression is a delegate invocation.

In the event that the primary expression is not a method group or a value of a delegate type, a compiler error is generated.

Element Access

An element access is made up of:

* + A primary-no-array-creation-expression, followed by
  + A “[“ token, followed by
  + An expression list, followed by
  + A “]” token.

The expression list consists of one or more expressions, which are separated by commas:

element-access:

primary-no-array-creation-expression [ expression-list ]

expression-list: expression

expression-list , expression

Array Access

For any array access, the primary-no-array-creation-expression of the element access will always be a value that is an array type.

The number of expressions in the expression list has to be the same as the rank of the array type. Each expression has to be of the type:

* int
* uint
* long
* ulong
* Any type that can be implicitly converted to one or more of the preceding types The result of evaluating an array access is a variable of the element type of the array.

Indexer Access

When dealing with indexer access, the primary-no-array-creation-expression of the element access will be one of the following:

* An interface type
* A struct
* A variable
* A value of a class

This Access

A this-access is made up of the reserved word this:

this-access: this

A this-access is only allowed in a code block of one of the following:

* An instance constructor
* An instance method
* An instance accessor

Base Access

A base-access is made up of the reserved word base followed by either:

* The “.” token and an identifier and optional type argument list

Or:

* + An expression list enclosed in square brackets The syntax is as follows:

base-access:

base . identifier type-argument-listopt base [ expression-list ]

new Operator

The new operator is used to create new instances of types. The new expression can take on three forms:

* + **Object-creation expressions.** Used to create new instances of class types and value types
  + **Array-creation expressions.** Used to create new instances of array types
  + **Delegate-creation expressions.** Used to create new instances of delegate types

While the new operator creates a new instance of a type, it does not mean that memory has been allo- cated, as this is handled automatically by the .NET Framework and will only consume resources when they are required.

typeof Operator

The typeof operator is used to obtain the System.Type object for a type:

typeof-expression: typeof ( type )

typeof ( unbound-type-name ) typeof ( void )

unbound-type-name:

identifier generic-dimension-specifieropt

identifier :: identifier generic-dimension-specifieropt unbound-type-name . identifier generic-dimension-specifieropt

generic-dimension-specifier:

< commasopt >

commas:

,

commas ,

sizeof Operator

The sizeof operator is used to return the number of 8-bit bytes occupied by a variable:

sizeof-expression:

sizeof ( unmanaged-type )

For many predefined types, the sizeof operator results in a constant int value, as shown in the follow- ing table:

|  |  |
| --- | --- |
| **Expression** | **Value** |
| sizeof(bool) | 1 |
| sizeof(byte) | 1 |
| sizeof(char) | 2 |
| sizeof(decimal) | 16 |
| sizeof(double) | 8 |
| sizeof(float) | 4 |
| sizeof(int) | 4 |
| sizeof(long) | 8 |
| sizeof(sbyte) | 1 |
| sizeof(short) | 2 |
| sizeof(uint) | 4 |
| sizeof(ulong) | 8 |
| sizeof(ushort) | 2 |

checked/unchecked Operators

The checked and unchecked operators are used to set the overflow-checking for integral-type arith- metic operations and conversions:

checked-expression: checked ( expression )

unchecked-expression: unchecked ( expression )

The checked operator is used to evaluate the contained expression in a checked context. The unchecked

operator, on the other hand, evaluates the contained expression in an unchecked context.

Default Value Expression

A default value expression obtains the default value of a type. Default value expressions are usually used to type parameters to work out whether they are value types or reference types:

default-value-expression: default ( type )

The result at runtime for reference values will be null, while if it is a value type, the result will be the default value of the type.

Anonymous Methods

An anonymous-method-expression is used to define anonymous methods. They evaluate to a value referencing the method:

anonymous-method-expression:

delegate anonymous-method-signatureopt block

anonymous-method-signature:

( anonymous-method-parameter-listopt ) anonymous-method-parameter-list:

anonymous-method-parameter

anonymous-method-parameter-list , anonymous-method-parameter

anonymous-method-parameter:

parameter-modifieropt type identifier

##### Unary Expressions

The following is a list of unary expressions:

unary-expression: primary-expression

+ unary-expression

- unary-expression

! unary-expression

~ unary-expression

pre-increment-expression pre-decrement-expression cast-expression

##### Cast Expressions

A cast-expression is used to explicitly convert an expression to a given type:

cast-expression:

( type ) unary-expression

##### Arithmetic Operators

The following operators are called the arithmetic operators:

* + \*
  + /
  + %

❑ +

* + – 131

The syntax of these expressions is as follows:

multiplicative-expression: unary-expression

multiplicative-expression \* unary-expression multiplicative-expression / unary-expression multiplicative-expression % unary-expression

additive-expression: multiplicative-expression

additive-expression + multiplicative-expression additive-expression – multiplicative-expression

##### Shift Operators

The two shift operators (<< and >>) are used to perform bit-shifting operations:

shift-expression: additive-expression

shift-expression << additive-expression

shift-expression right-shift additive-expression

The << operator shifts a value left by a number of bits specified, while the >> operator shifts a value right by a number of bits specified.

##### Relational/Type Testing Operators

Six relational and type-testing operators are available in C#:

❑ ==

* !=
* <
* >
* <=
* >=

The syntax of these is as follows:

relational-expression: shift-expression

relational-expression < shift-expression relational-expression > shift-expression relational-expression <= shift-expression relational-expression >= shift-expression relational-expression is type

relational-expression as type

equality-expression: relational-expression

equality-expression == relational-expression equality-expression != relational-expression

These are all comparison operators. All predefined comparison operators return a result of the bool type. The following table lists operators, along with the outcome of the operator on operands:

**Operator Outcome**

x == y x != y x < y

x > y x <= y x >= y

If x is equal to y, the result is true.

If x is not equal to y, the result is false.

If x is equal to y, the result is false.

If x is not equal to y, then the result is true.

If x is less than y, the result is true.

If x is greater than y, the result is false.

If x is less than y, the result is false.

If x is greater than y, the result is true.

If x is less than or equal to y, the result is true.

If x is greater than or equal to y, the result is false.

If x is less than or equal to y, the result is false.

If x is greater than or equal to y, the result is true.

##### Logical Operators

Three logical operators are available in C#:

* + &
  + |
  + ^

The & operator computes the bitwise logical AND of the two operands. The logical AND operation com- pares 2 bits, and if they are both “1”, the result is “1”; otherwise, the result is “0”.

The | operator computes the bitwise logical OR of the two operands. The logical OR operation compares 2 bits, and if they are both “1”, the result is “1”; otherwise, the result is “0”.

The ^ operator computes the bitwise logical exclusive OR of the two operands. The logical exclusive OR (XOR) operation compares 2 bits, and if exactly one of them is “1” (that is, if they are different values), the result is “1”; otherwise (if the bits are the same), the result is “0”.

##### Conditional Logical Operators

There are two logical conditional operators in C#:

* + &&
  + || 133

The following is the syntax for these operators:

conditional-and-expression: inclusive-or-expression

conditional-and-expression && inclusive-or-expression

conditional-or-expression: conditional-and-expression

conditional-or-expression || conditional-and-expression

The simplest way to think of && and || is as conditional forms of & and |. What do we mean by that? Well, let’s look at the following operations:

x && y x || y

These are equivalent to these operations:

x & y x | y

The only difference is that y in:

x && y

is evaluated only if x is true, while for:

x || y

y is evaluated only is x is false.

##### Null Coalescing Operator

The ?? operator is called a null coalescing operator:

null-coalescing-expression: conditional-or-expression

conditional-or-expression ?? null-coalescing-expression

The ?? operator allows conditional expressions to be written that are an excellent shorthand way of replacing if statements. They take on the form:

b ? x : y

First, the condition b is evaluated. If b is true, x is evaluated and becomes the result of the operation; otherwise, y is evaluated and this becomes the result of the operation.

A conditional expression can never evaluate x and y.

##### Assignment Operators

The assignment operators are used to assign a new value to a variable, event, property, or indexer element. Eleven assignment operators are available in C# (most of these you will have come across already):

❑ =

❑ +=

* -=
* \*=
* /=
* %=
* &=
* |=
* ^=
* <<=
* >>=

The = operator is called a simple assignment operator. It is used to assign the value of the right operand to the variable, property, or indexer element given by the left operand.

The operators created by prefixing an = character with a binary operator are called the compound assignment operators. These operators carry out operations on the two operands and then assign the resulting value to the variable, property, or indexer element given by the left operand.

The += and -= operators with an event access expression as the left operand are called the event assign- ment operators.

##### Expression

An expression is either a conditional-expression or an assignment:

expression:

conditional-expression assignment

##### Constant Expressions

A constant expression can be fully and completely evaluated at the point that the code is compiled:

constant-expression: expression

A constant expression can have any one of the following types:

* bool
* byte
* char
* decimal
* double
* enumeration type
* float
* int
* long
* null type
* sbyte
* short
* string
* uint
* ulong
* ushort

For more information on these types, check out Chapter 6.

The following constructs are all allowed in constant expressions:

* Literals
* Null literals
* References to const members of class and struct types
* References to members of enumeration types
* Cast expressions (as long as the type is one of the following: bool, byte, char, decimal, double, enumeration type, float, int, long, null type, sbyte, short, string, uint, ulong, or ushort)
* The following unary operators:

❑ +

* + –
  + !
  + ~
    - The following binary operators:

❑ +

* + - * –
      * \*
      * /
      * %
      * <<
      * >>
      * &
      * |
      * ^
      * &&
      * ||

❑ ==

* + - * !=
      * <
      * >
      * <=
      * >=
    - As long as each operand is one of the following:
      * bool
      * byte
      * char
      * decimal
      * double
      * enumeration type
      * float
      * int
      * long
      * null type
      * sbyte
      * short
  + string
  + uint
  + ulong
  + ushort
* The ?: operator
* sizeof expressions

##### Boolean Expressions

All Boolean expressions will return a result of the type bool:

boolean-expression: expression

The bool type has two possible values:

* true
* false

Boolean expressions are important in a number of other C# statements where a controlling conditional statement is required. These statements are:

* Do
* For
* If
* While

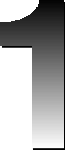
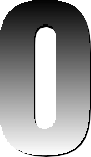
Boolean expressions have to be of a type that can be implicitly converted to bool or that implements

operator true.

##### Summary

In this chapter we’ve taken a detailed look at expressions in C#. These expressions will form the back- bone of a majority of code that a programmer will create.

In Chapter 10, you look at C# statements.

### Statements

Statements are everywhere in code. Nearly every line that you write is going to be a statement. Statements are a way to take your thoughts and organize them into logical code that the compiler can follow and process. A good understanding of statements in C# is essential to being able to write good code.

##### What are Statements?

A statement in C# (or almost every other programming language going) can be thought of as equiv- alent to a complete sentence in the English language. It might seem odd to compare a programming language with a real, living language, but this happens to be the best and easiest analogy.

For example, if someone says:

I like C#.

You know exactly what they mean. However, if they said:

I like.

or

I C#.

You would realize that there’s something wrong with these sentences. They’re not complete, and they are ambiguous.

The same is true for a statement in C#. A statement in C# is a complete instruction that the com- piler understands and can process. The statement has to be valid and make sense to the compiler, and it has to follow syntax rules just as sentences in English must.

Here’s a simple statement in C#:

var1 = 3 + 4;

This is a single statement in C#. It’s logical and makes perfect sense to the compiler, which will take the two numbers, add them together, and store the result in a variable called var1.

No ambiguities. No problems.

C# statements don’t end with a period like sentences in English but instead with a semicolon (;). This is used to indicate to the compiler that the statement has ended. Just as sentences in English don’t make any sense if the period is missing and they run into one another, C# statements that don’t have the termi- nator at the end are also not valid.

Just as sentences build on one another to form paragraphs, statements build to form code blocks. In code blocks, statements are processed one by one:

{

statement1; statement2; statement3;

}

In this code block, three statements are processed one after the other, starting with statement1 and ending after statement3.

There would be nothing technically wrong with putting all the statements on a single line — the com- piler can still find the end of each statement because of the semicolon:

{

statement1;statement2;statement3;

}

The problem with this kind of layout is that it makes reading the code and future debugging an awful experience.

The following layout is looser and makes it easier to read the code:

{

statement1; statement2; statement3;

}

So far, all this seems simple enough, but as you can imagine, there are numerous specific rules governing statements, and we will be looking at these rules in the remainder of this chapter.

##### C# Statements

A number of different types of statements are possible in C#:

statement:

labeled-statement declaration-statement embedded-statement

embedded-statement: block

empty-statement expression-statement selection-statement iteration-statement jump-statement

try-statement checked-statement unchecked-statement lock-statement using-statement yield-statement

An embedded-statement is used within other statements, and these must be placed within code blocks.

This is a valid embedded-statement:

public class Test

{

public static void Main()

{

bool i = false;

if ( i)

{

int j = 7;

}

}

}

While this is invalid:

public class Test

{

public static void Main()

{

bool i = false;

if ( i) int j = 7;

}

}

End Point and Reachability

There are two other concepts that a programmer needs to be comfortable with:

* End point
* Reachability

End Point

Every valid statement has an end point. The end point of a statement is the end of the statement itself. Embedded statements within statements are called *composite statements*.

Reachability

If a statement can be reached during code execution, this statement is said to be *reachable*. If that state- ment cannot be reached, it is said to be *unreachable*.

The following code contains reachable and unreachable statements:

public class Test

{

public static void Main()

{

int x = 6;

const int y = 7; if ( x == 6)

System.Console.WriteLine(“Reachable”); if ( y == 6) System.Console.WriteLine(“Unreachable”);

}

}

What makes the unreachable statement unreachable? It’s that the value of y is defined as a constant and as such cannot change. This is detected by the compiler, and a warning is issued:

C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727>csc test.cs Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42 for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727

Copyright (C) Microsoft Corporation 2001-2005. All rights reserved. test.cs(10,9): warning CS0162: Unreachable code detected C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727>

In this example, both statements are reachable:

public class Test

{

public static void Main()

{

int x = 6;

const int y = 6; if ( x == 6)

System.Console.WriteLine(“Reachable”); if ( y == 6) System.Console.WriteLine(“Reachable”);

}

}

Take a look at this code, a slight variation of the preceding code:

public class Test

{

public static void Main()

{

int x = 7;

const int y = 7;

if ( x == 6)

System.Console.WriteLine(“Potentially reachable”);

if ( y == 6) System.Console.WriteLine(“Unreachable”);

}

}

Even though the value of x makes the if statement that refers to it currently unreachable, it is poten- tially reachable because the value of x could later be changed.

If the unreachable statement is removed (or modified) to make it reachable, the potentially reachable statement now generates a warning.

public class Test

{

public static void Main()

{

int x = 7;

const int y = 7; if ( x == 6)

System.Console.WriteLine(“Potentially reachable”); if ( y == 7) System.Console.WriteLine(“Unreachable”);

}

}

The following is always considered reachable:

* + The block of a function member
  + The block of an anonymous-method-expression

Reachability is determined by the compiler by evaluating each statement in a block. By carrying out this operation successively, the reachability of any statement can be determined.

There are two scenarios where a compile-time error is generated when the end point of a statement is reachable:

* If the end point of a function that computes a value is reachable. In this case, the return state- ment is usually missing.
* If the end point of the statement list of a switch section is reachable. This is usually the case when a break statement is missing.

##### Code Blocks

A code block (also called a block) is a way to allow multiple statements to be written in situations where only a single statement is allowed.

block:

{ statement-listopt }

A code block consists of an optional statement-list. This is enclosed in braces ({ and }). If the state- ment list is omitted, the code block is said to be empty.

A block can also contain declaration statements, and the scope of a local variable or constant declared in a code block is the block itself and no more.

A block of code is executed as follows:

* If the code block is empty, control is passed straight to the end point of the code block.
* If the block contains statements, control is transferred to the statement list, and the statements are executed. If control reaches the end point of the statement list, control is transferred to the end point of the code block.

The statement list of a code block is always reachable if the block is reachable.

Statement Lists

A statement list consists of one or more statements written and presented in a sequence. Statement lists can be found in code blocks or in switch blocks.

statement-list: statement

statement-list statement

Statement lists are executed when the control is transferred to the first statement in the list. If control reaches the end of the statement in the list, control is transferred to the end point of the statement list.

For a statement in a statement list to be reachable, the following have to be true:

* The statement is the first in the statement list, and the statement list is reachable (the first state- ment in any reachable statement list is reachable).
* The end point of the statement coming before the current statement is reachable.
* The statement is labeled, and the label is referenced by a goto statement that is itself reachable.

For the end point of a statement list to be reachable, the end point of the last statement in the list also has to be reachable.

##### Empty Statements

An empty statement does nothing. It is used when there are no operations to perform but a statement is required (such as in a while statement).

empty-statement:

;

When executed, an empty statement merely transfers control to the end point of the statement. The end point of an empty statement is always reachable.

##### Labeled Statements

A labeled statement has been prefixed by a label. This label is used to declare a unique name for the statement. These labeled statements are referenced from goto statements:

labeled-statement: identifier : statement

The scope of a label is limited to the block where the label is declared (this includes any nested blocks that the main block contains).

class Test

{

static void Main() { goto X;

X: Console.Write(“Hello, World!”);

}

}

No two labels that share the same scope can have the same name without causing a compiler error, as will happen when compiling the following example:

class Test

{

static void Main() { goto X;

X: Console.Write(“Hello, “);

X: Console.Write(“World!”);

}

}

*Note that label names don’t interfere with other identifiers in code. This means that you could have a label, a variable, and a parameter all with the same name in the same block of code.*

A labeled statement is reachable if the label is referenced by a goto statement that is itself reachable. The only exception is where the goto statement is inside a try that includes a finally block whose end point is unreachable, and the labeled statement is outside the try.

##### Declaration Statements

Declaration statements are used to declare either a local variable or a constant. Declaration statements are allowed inside code blocks, but they are not allowed inside any embedded statements:

declaration-statement:

local-variable-declaration ; local-constant-declaration ;

Local Variable Declarations

Local variable declarations are used to declare one or more local variables:

local-variable-declaration:

type local-variable-declarators

local-variable-declarators: local-variable-declarator

local-variable-declarators , local-variable-declarator

local-variable-declarator: identifier

identifier = local-variable-initializer

local-variable-initializer: expression

array-initializer

The type of declaration specifies the type of the variables brought into existence by the declaration.

The type is followed by a list of declarators, each of which specifies a new variable. A declarator consists of an identifier that names the variable and is optionally followed by an = token and an initializer that gives the initial value of the variable.

The value of a local variable is retrieved by an expression using a simple name, while the value of a local variable is modified using an assignment. A local variable has to be definitely assigned at each location where its value is retrieved.

The scope of a local variable declared in a local variable declaration is the block in which the declaration is found. Code cannot refer to a local variable in a textual position that comes before the local variable declarator of the local variable.

Also, you cannot declare another variable or constant within the scope of another variable or constant with the same name.

Here are two ways to declare and assign a variable:

class Test

}

And:

{

static void Main() { int x = 7;

}

class Test

{

static void Main() { int x;

x = 7;

}

}

The code in these two blocks is functionally equivalent.

Local Constant Declarations

Local constant declarations are used to declare one or more local constants:

local-constant-declaration:

const type constant-declarators

constant-declarators: constant-declarator

constant-declarators , constant-declarator

constant-declarator:

identifier = constant-expression

The type of declaration specifies the type of the constants brought into existence by the declaration.

The type is followed by a list of declarators, each of which specifies a new constant. A declarator consists of an identifier that names the variable and is optionally followed by an = token and an initializer that gives the initial value of the constant.

The value of a local constant is retrieved by an expression using a simple name.

The scope of a local constant declared in a local constant declaration is the block in which the declaration is found.

Also, you cannot declare a constant within the scope of another constant with the same name.

##### Expression Statements

Expression statements are used to evaluate an expression. Values that result from expressions are discarded unless they are preserved (by assigning them to variables):

expression-statement: statement-expression ;

statement-expression: invocation-expression object-creation-expression assignment

post-increment-expression post-decrement-expression pre-increment-expression pre-decrement-expression

It is important to note that some expressions are not permitted. For example, the following are used only to compute values and are not in themselves valid expressions:

x + y + z; x ==7;

Execution of an expression statement evaluates the expression and, after that is completed, transfers con- trol to the end point of the expression statement.

The end point of an expression statement is always reachable if that expression statement itself is reachable.

Selection Statements

Selection statements are used to select appropriate statements to run from a list of possible statements. The decision as to what statements to run is based on the outcome of a selection expression:

selection-statement: if-statement switch-statement

The if Statement

The if statement is used to select statements for execution based on the value of a Boolean expression:

if-statement:

if ( boolean-expression ) embedded-statement

if ( boolean-expression ) embedded-statement else embedded-statement

The if statement also allows for there to be an else clause. The else clause is associated with the lexi- cally nearest preceding if allowed by the syntax.

The following code examples show equivalent if statements:

if (x)

{

if (y)

{ A();

}

else

}

{ B();

}

And:

if (x) if (y) A(); else B();

Which style you use is a personal choice.

The steps carried out to execute an if statement are as follows:

* + The Boolean expression that the if statement depends on is first evaluated.
  + If the Boolean expression evaluates to true, control is transferred to the first embedded statement. If control reaches the end point of that statement, control is transferred to the end point of the entire if statement.
  + If the Boolean expression evaluates to false and an else clause is present, control is transferred to the second embedded statement. If control reaches the end point of that statement, control is transferred to the end point of the if statement.
  + If the Boolean expression evaluates to false and if an else clause is not specified, control is transferred to the end point of the if statement.

The first embedded statement of any if statement will be reachable if the if statement is reachable and the Boolean expression does not have the constant value false.

The second embedded statement of an if statement, if present, will be reachable if the if statement is reachable and the Boolean expression does not have the constant value true.

The end point of any if statement will be reachable if the end point of at least one of the embedded statements is reachable. The end point of an if statement with no else part will be reachable if the if statement is reachable and the Boolean expression does not have the constant value true.

The switch Statement

The switch statement selects a statement list for execution that has a switch label that corresponds to the value of the switch expression.

A switch statement is a substitute for multiple if statements — both work in the same way. It is ulti- mately a matter of style as to which of them to use. The syntax is as follows:

switch-statement:

switch ( expression ) switch-block

switch-block:

{ switch-sectionsopt }

switch-sections:

switch-section

switch-sections switch-section

switch-section:

switch-labels statement-list

switch-labels:

switch-label

switch-labels switch-label

switch-label:

case constant-expression :

default :

The switch statement consists of four parts:

* At the core of the switch statement is the keyword switch.
* Following this keyword is a parenthesized expression called the switch expression.
* This is followed by a switch block. A switch block is made up of zero or more switch sections enclosed in braces.
* Switch sections are made up of one or more switch labels followed by a statement list.

Here is an example of a switch statement. We have labeled which statement is executed with the words “executed.”

public class test

{

public static void Main()

{

test a = new test(); a.xyz(1);

}

void xyz(int i)

{

switch (i)

{

case 0:

System.Console.WriteLine(“not executed”); break;

case 1: System.Console.WriteLine(“executed”); break;

default:

System.Console.WriteLine(“not executed”); break;

}

}

}

The governing type of a switch statement is worked out by the switch expression. If the type of the

switch expression is any of the following types, that will become the governing type:

* + byte
  + sbyte
  + char
  + int
  + uint
  + long
  + ulong
  + short
  + ushort
  + string
  + an enum type

Otherwise, one (and one only) user-defined implicit conversion operator will be present that will convert from the type of the switch expression or a base type of this type to one of the following governing types:

* + byte
  + sbyte
  + char
  + int
  + uint
  + long
  + ulong
  + short
  + ushort
  + string

If no implicit conversion operator exists or if more than one such implicit conversion operator is present, a compiler error will be generated.

Switch statements are executed as follows:

* + The switch expression is evaluated and converted to the appropriate governing type.
  + If one of the constants specified in a case label in the same switch statement matches the value of the switch expression, control is transferred to the statement list that follows the matched case label.
* If none of the constants specified in case labels in the same switch statement is equal to the value of the switch expression and if a default label is present, control is then transferred to the statement list that follows the default label.
* If none of the constants specified in case labels in the same switch statement is equal to the value of the switch expression and no default label is present, control is transferred to the end point of the switch statement.

In the following code example, the statement list after the default label is run:

public class test

{

public static void Main()

{

test a = new test();

a.xyz(7);

}

void xyz(int i)

{

switch (i)

{

case 0:

System.Console.WriteLine(“not executed”); break;

case 1:

System.Console.WriteLine(“not executed”); break;

default: System.Console.WriteLine(“executed”); break;

}

}

}

Note that statement lists in a switch section usually end with one of the following statements:

* break
* goto case
* goto default

However, any statement that makes the end point of the list unreachable is valid (for example, a while

statement controlled by a Boolean expression that evaluates to true).

Multiple labels are allowed in switch sections:

public class test

{

public static void Main()

{

test a = new test();

a.xyz(2);

}

void xyz(int i)

{

switch (i)

{

case 0:

System.Console.WriteLine(“not executed”); break;

case 1:

System.Console.WriteLine(“not executed”); break;

case 2:

default: System.Console.WriteLine(“executed”); break;

}

}

}

The statement lists contained in a switch block are allowed to contain declaration statements. The scope of these local variables or constants will be the switch block in which they are declared.

The statement list of a given switch section is reachable if the switch statement is reachable and if one or more of the following are true:

* + The switch expression is a constant value that matches a case label in the switch section.
  + The switch expression is a nonconstant value.
  + The switch expression is a constant value that doesn’t match any case label, but the switch

section contains the default label.

* + A switch label of the switch section is referenced by a goto case or goto default statement that is itself reachable.

The end point of a switch statement is reachable if one or more of the following are true:

* + The switch statement contains a reachable break statement that exits the switch statement.
  + The switch statement is reachable, the switch expression is a nonconstant value, and there is no default label present.
  + The switch statement is reachable, the switch expression is a constant value that doesn’t match any case label, and no default label is present.

Iteration Statements

Iteration statements are used to execute an embedded statement repeatedly:

iteration-statement: while-statement do-statement for-statement

foreach-statement

The while Statement

The while statement is used to conditionally execute an embedded statement zero or more times:

while-statement:

while ( boolean-expression ) embedded-statement

All while statements are evaluated as follows:

* First, the Boolean expression is evaluated.
* If the Boolean expression evaluates to true, control is transferred to the embedded statement. If control reaches the end point of the embedded statement, control is transferred to the beginning of the while statement.
* If the Boolean expression evaluates to false, control is transferred to the end point of the while

statement.

The embedded statement of a while statement is reachable when the while statement is reachable and the Boolean expression is not set to have the constant value false.

The end point of a while statement will be reachable if at least one of the following is true:

* The while statement contains a reachable break statement that exits the while statement.
* The while statement is reachable, and the Boolean expression is not set to have the constant value true.

The do Statement

The do statement is used to conditionally execute an embedded statement one (not zero) or more times:

do-statement:

do embedded-statement while ( boolean-expression ) ;

All do statements are executed as follows:

* Control is initially passed to the embedded statement.
* If control reaches the end point of the embedded statement, the Boolean expression is evaluated. If that Boolean expression evaluates to true, control is transferred to the beginning of the do statement, and another iteration cycle is processed. If the Boolean expression evaluates to false, control is transferred to the end point of the do statement.

The embedded statement of a do statement is always reachable if the do statement itself is reachable. The end point of a do statement will be reachable if at least one of the following is true:

* + The do statement contains a reachable break statement that exits the do statement.
  + The end point of the embedded statement is reachable, and the Boolean expression does not have the constant value true.

The for Statement

The for statement is used to evaluate a sequence of initialization expressions. While the condition eval- uates to true, the for statement repeatedly executes the statement and each time evaluates the iteration expressions.

for-statement:

for ( for-initializeropt ; for-conditionopt ; for-iteratoropt ) embedded-statement for-initializer:

local-variable-declaration statement-expression-list

for-condition:

boolean-expression

for-iterator:

statement-expression-list

statement-expression-list:

statement-expression

statement-expression-list , statement-expression

A for statement is executed as follows:

* + If a for initializer is present, the variable initializers or statement expressions are executed in the order they are written. This step is only carried out once, no matter how many times the statement is executed.
  + If a for condition is present, it is next evaluated.
  + If the for condition is not present or if the evaluation evaluates to true, control is transferred to the embedded statement. If control reaches the end point of the embedded statement, the expressions of the for iterator, if any, are evaluated in sequence, and then another iteration is performed, starting with evaluation of the for condition from the preceding step.
  + If the for condition is present and the evaluation evaluates to false, control is then transferred to the end point of the for statement.

The embedded statement of a for statement is reachable if one of the following is true:

* + The for statement is reachable, and so no for condition is present.
  + The for statement is reachable, and a for condition is present but does not have the constant value false.

The end point of a for statement will be reachable if at least one of the following is true:

* The for statement contains a reachable break statement that exits the for statement.
* The for statement is reachable and a for condition is present but does not have the constant value true.

The foreach Statement

The foreach statement enumerates the elements of a collection, executing an embedded statement for each element of the collection:

foreach-statement:

foreach ( type identifier in expression ) embedded-statement

The type and identifier of a foreach statement declare the iteration variable of the statement. The itera- tion variable is a read-only local variable that has scope that extends over the embedded statement.

When the statement is executed, the iteration variable is used to represent the collection element for which an iteration is currently being performed.

A compiler error is generated if the embedded statement tries to modify the iteration variable in any way or if an attempt is made to pass the iteration variable as a ref or out parameter.

Jump Statements

Jump statements are used to unconditionally transfer control to another statement in the code. The loca- tion to which the jump occurs is called the target of the jump statement:

jump-statement: break-statement

continue-statement goto-statement return-statement throw-statement

Jump statements can transfer control from a block of code but not into a block of code.

The break Statement

The break statement is used to exit an enclosing do, for, foreach, switch, or while statements (in fact, break statements have to be enclosed by one of these statements or a compiler error will occur):

break-statement: break ;

In the event that a break statement is enclosed in a nested set of statements, the break statement applies only to the innermost statement.

All break statements are processed as follows:

* If the break statement is used to exit one or more try blocks that have associated finally blocks, control is first transferred to the finally block of the innermost try statement. If con- trol reaches the end point of a finally block, control is then transferred to the finally block

of the next enclosing try statement. This process is repeated until the finally blocks of all try

statements have been executed.

* + Control is then transferred to the target of the break statement. The end point of a break statement is never reachable.

The continue Statement

The continue statement is used to begin a new iteration cycle of the enclosing do, for, foreach, and

while statements:

continue-statement:

continue ;

When there are multiple enclosing do, for, foreach, and while statements, the continue statement only applies to the innermost enclosing statement.

The end point of the continue statement is never reachable. A continue statement is processed as follows:

* + If the continue statement is used to exit one or more try blocks with associated finally blocks, control is first passed to the finally block of the innermost try statement. If control reaches the end point of a finally block, control is then passed to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all try statements have been executed.
  + Control is transferred to the target of the continue statement.

The goto Statement

The goto statement is used to transfer control to a statement that has been marked using a label:

goto-statement:

goto identifier ;

goto case constant-expression ; goto default ;

The target of any goto identifier statement is a statement marked by a label. If a label with the given name does not exist in the current function member, or if the goto statement is not within the scope of the label, a compiler error is generated.

A goto statement is executed as follows:

* + If the goto statement is used to exit one or more try blocks with associated finally blocks, control is first passed to the finally block of the innermost try statement. If control reaches the end point of a finally block, control is then transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all try statements have been executed.
  + Control is transferred to the target of the goto statement.

The end point of a goto statement is always unreachable.

The return Statement

The return statement is used to return control to the caller of the function member:

return-statement:

return expressionopt ;

A return statement is executed as follows:

* If the return statement is used to specify an expression, the expression is evaluated and the resulting value is converted to the return type of the containing function member using an implicit conversion. The result of the conversion is then set as the value returned to the caller.
* If the return statement is enclosed by one (or more) try blocks that have finally blocks, con- trol is first passed to the finally block of the innermost try statement. If control reaches the end point of a finally block, control is then transferred to the finally block of the next enclosing try statement. This process is repeated until all the finally blocks of all enclosing try statements have been executed.
* Control is returned to the caller of the containing function member. The end point of a return statement is always unreachable.

The throw Statement

The throw statement is used to throw exceptions:

throw-statement:

throw expressionopt ;

A throw statement with an expression is used to throw the value produced by evaluating the expres- sion. The expression will indicate a value of the class type System.Exception or a class type derived from System.Exception. If, on evaluation, the expression results in a null, a System.NullReferenceException will be thrown instead.

The throw statement can be used with expressions that have a type given by a type parameter only where that type parameter has System.Exception or a subclass of System.Exception as the effective base class.

A throw statement with no expression can only be used in catch blocks. Here the statement will rethrow the exception currently being handled by that catch block.

The end point of a throw statement is always unreachable.

The using Statement

The using statement is used to obtain one or more resources, execute a statement, and finally dispose of the resources:

using-statement:

using ( resource-acquisition ) embedded-statement

resource-acquisition:

local-variable-declaration expression

A resource is a class or struct that implements the System.IDisposable interface.

The using statement is only useful for objects with a lifetime that does not extend beyond the method in which the objects are constructed.

A using statement is translated into three parts:

* + Acquisition
  + Usage
  + Disposal

Usage of the resource will be implicitly enclosed in a try statement that includes a finally clause. This

finally clause is used to dispose of the resource when it is finished.

Instantiated objects must implement the System.IDisposable interface. Note that the following code snippets are equivalent in function:

using (ResourceType resource = expression) embedded-statement

And:

{

ResourceType resource = expression; try

{

embedded-statement

}

finally

{

}

}

The yield Statement

The yield statement is used inside iterator blocks to yield a value to the enumerator object. It is also used to indicate the end of the iteration.

yield-statement:

yield return expression ; yield break ;

*Note that in order to maintain compatibility,* yield *is not a keyword. Instead, it has special meaning only when it is used before a* return *or* break *keyword. In all other contexts,* yield *is used as an identifier.*

There are a number of restrictions on the location where a yield statement can appear.

* A yield statement cannot appear outside any of the following: accessor-body, method-body, or operator-body.
* A yield statement cannot appear anywhere in a try statement that contains catch clauses.
* A yield statement cannot appear in the finally clause of a try statement.
* A yield statement cannot appear inside an anonymous method. A yield return statement is executed as follows:
* The expression that appears in the statement is evaluated and implicitly converted to the yield

type. This is assigned to the Current property of the enumerator object.

* Execution of the iterator block is halted. If the yield return statement is within one or more

try blocks, the associated finally blocks are not yet executed.

* The MoveNext method of the enumerator object returns true to the caller. This indicates that the enumerator object has moved on to the next item.

A yield break statement is executed as follows:

* If the yield break statement is enclosed by one or more try blocks that have finally blocks, control is first transferred to the finally block of the innermost try statement. If control reaches the end point of a finally block, control is then passed to the finally block of the next enclosing try statement. This process is looped until the finally blocks of all enclosing try statements have been executed.
* Control is then returned to the caller of the iterator block. This is either the MoveNext method or

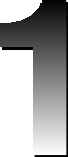
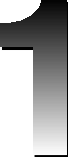
Dispose method of the enumerator object.

The end point of a yield break statement is always unreachable.

##### Summary

In this chapter you looked at C# statements. The chapter started off by taking a broad look at statements and how they work, before taking at look at specific statements present in C#.

In Chapter 11, you look at namespaces and how they are used in C#.

### Namespaces

In this chapter you examine how namespaces are used in C# code to organize programs.

##### What are Namespaces?

Namespaces are used extensively in C# programs. There are two ways that namespaces are used:

* + To organize classes in the .NET Framework
  + To declare namespaces, which can help control the scope of class and method names used

This can be done through either internal organization (organizing the internal structure of the program itself) or external organization (controlling how program elements are exposed to other programs).

*Namespaces DO NOT correspond to file or folder names used to store source code. However, if naming folders and files to correspond to namespaces helps you organize your code, you are free to do so; just remember that it is not a requirement.*

Organizing Classes

By using namespaces in code, a programmer can have the luxury of writing less code because namespace identifiers do not have to be used. In addition, namespaces reduce conflicts with other libraries and at the same time offer code that is more readable.

Take the following example:

System.Console.WriteLine(“Hello, “); System.Console.WriteLine(“World!”);

Using the using keyword means that the entire name is not required:

using System;

Console.WriteLine(“Hello, “); Console.WriteLine(“World!”);

Controlling Scope

Here’s a simple example that shows how namespaces can be used to control the scope of class and method names:

namespace MyNamespace

{

class MyClass

{

public void MyMethod()

{

System.Console.WriteLine(

“MyMethod contained inside MyNamespace”);

}

}

}

Let’s now take a closer look at namespaces, beginning with compilation units.

##### Compilation Units

Compilation units define the C# source file. A compilation unit consists of:

* Zero or more extern-alias-directives, followed by
* Zero or more using-directives, followed by
* Zero or more global-attributes, followed by
* Zero or more namespace-member-declarations:

compilation-unit:

extern-alias-directivesopt using-directivesopt global-attributesopt namespace-member-declarationsopt

A C# program is made up of one or more compilation units. Each of these compilation units corresponds to a separate C# source file. When the final C# program is compiled, the compilation units are all processed.

The extern-alias-directives of a compilation unit affect the using-directives, global- attributes, and namespace-member-declarations of that particular compilation unit. They have no effect on other compilation units.

The using-directives of a compilation unit affect the global-attributes and namespace-member- declarations of that compilation unit. They have no effect on other compilation units.

The global-attributes of a compilation unit allow the specification of attributes for the target assem- bly. Assemblies act as physical containers for types.

The namespace-member-declarations of each compilation unit of a program supply members to a single declaration space called the global namespace.

##### Namespace Declarations

A namespace declaration consists of:

* + The keyword namespace, followed by
  + A namespace name and body, optionally followed by
  + A semicolon:

namespace-declaration:

namespace qualified-identifier namespace-body ;opt

qualified-identifier: identifier

qualified-identifier . identifier

namespace-body:

{ extern-alias-directivesopt using-directivesopt namespace-member-declarationsopt

}

A namespace declaration can occur either:

* + As a top-level declaration in a compilation-unit. Here the namespace becomes a member of the global namespace.
  + As a member declaration within another namespace-declaration.

Here the namespace-declaration occurs within another namespace-declaration; the inner namespace becomes a member of the outer namespace.

In both cases, the name of a namespace will be unique within the containing namespace.

It is important to note that namespaces are implicitly public and that the namespace declaration cannot include any access modifiers.

The optional using-directives import the names of other namespaces and types. This allows them to be referenced directly rather than through the use of qualified names.

The optional namespacemember-declarations contribute members to the declaration space of the namespace.

All extern-alias-directives have to be placed before any using-directives, and all extern- alias-directives and similarly all using-directives have to appear before any member declarations.

The qualified-identifier of a namespace-declaration can be a single identifier or a sequence of identifiers separated by “.” tokens. Using a sequence of identifiers allows a program to define a nested namespace without having to actually nest several namespace declarations. This means that the follow- ing lines of code are equivalent:

namespace NS1.NS2

{

class A {} class B {}

}

And:

namespace NS1

{

namespace NS2

{

class A {} class B {}

}

}

Namespaces are open-ended. This means that two namespace declarations with the same fully qualified name contribute to the same declaration space. Thus, the two code snippets that follow are equivalent:

namespace NS1.NS2

{

class A {}

}

namespace NS1.NS2

{

class B {}

}

And:

namespace NS1.NS2

{

class A {} class B {}

}

##### Extern Alias Directives

An extern-alias-directive is used to define an identifier that acts as an alias for an externally defined namespace.

The specification of the aliased namespace is external to the source code of the program:

extern-alias-directives: extern-alias-directive

extern-alias-directives extern-alias-directive

extern-alias-directive: extern alias identifier ;

The scope of an extern-alias-directive covers the following immediately containing compilation- unit or namespace-body:

* + using-directives:
  + global-attributes:
  + namespacemember-declarations

A type is always declared as a member of a single namespace. However, it is possible for a namespace hierarchy referenced by an extern alias to contain types that are also members of other namespaces.

##### Using Directives

Using directives are used to allow for the use of namespaces and types that are defined in other names- paces. In doing this, however, they do not contribute new members to the declaration spaces of the com- pilation units or namespaces where they are used. The syntax is as follows:

using-directives: using-directive

using-directives using-directive

using-directive:

using-alias-directive using-namespace-directive

There is a subtle difference between the using-alias-directive and using-namespace-directive:

* + A using-alias-directive introduces an alias for a namespace or type.
  + A using-namespace-directive imports the type members of a namespace.

Using Alias Directives

A using-alias-directive is used to define an identifier that acts as an alias for a namespace or type within the enclosing compilation unit or namespace body:

using-alias-directive:

using identifier = namespace-or-type-name ;

Using Namespace Directives

A using-namespace-directive is used to import types contained in a namespace into the immedi- ately enclosing compilation unit or namespace body. This allows the identifier of each type to be used without qualification:

using-namespace-directive: using namespace-name ;

Namespace Members

A namespace-member-declaration is either a:

* Namespace-declaration
* Type-declaration

The syntax is as follows:

namespace-member-declarations: namespace-member-declaration

namespace-member-declarations namespace-member-declaration

namespace-member-declaration: namespace-declaration type-declaration

Both compilation units or namespace bodies can contain namespace-member-declaration. This means that the namespace-member-declaration adds new members to the underlying declaration space of the compilation unit or namespace body.

Type Declarations

A type declaration is either a:

* class-declaration
* struct-declaration
* interface-declaration
* enum-declaration
* delegate-declaration

The syntax is as follows:

type-declaration: class-declaration struct-declaration

interface-declaration enum-declaration delegate-declaration

It is possible for a type declaration to occur as of the following:

* + Top-level declaration in a compilation unit
  + A member declaration within a namespace, class, or struct Here are the access modifiers for type declarations:
  + Types that have been declared as part of compilation units or namespace declarations can have either public or internal (default) access.
  + Types declared in classes can have public, protected internal, protected, internal, or private (default) access.
  + Types declared in structs can have public, internal, or private (default) access.

Qualified Alias Member

A qualified-alias-member provides explicit access to the global namespace and to extern or using

aliases that might be hidden and made inaccessible by other entities.

The syntax is as follows:

qualified-alias-member:

identifier :: identifier type-argument-listopt

A qualified-alias-member can be used as either of the following:

* + A namespace-or-type-name
  + As the left operand in a member-access

A qualified-alias-member consists of two identifiers:

* + Left-hand identifiers
  + Right-hand identifiers

These identifiers, described as follows, are separated by the :: token, and this is then optionally fol- lowed by a type-argument-list.

When the left-hand identifier is global, the global namespace is examined for the right-hand identifier. For any other left-hand identifier, that identifier is looked up as an extern or using alias.

*A compile-time error results if there is no such alias or the alias references a type.*

There are two forms that a qualified-alias-member can take:

* + A::B<G1, ..., GN>

Here A and B are used to represent identifiers, and <G1, ..., GN> is a type argument list.

* + A::B

Here A and B again represent identifiers.

Here is how the meaning of a qualified-alias-member is worked out:

* If A is the identifier global, the global namespace is searched for B:
  + If the global namespace contains a namespace named B and N is zero, the qualified- alias-member will refer to that namespace.
  + If the global namespace contains a non-generic type named B and N is zero, the

qualified-alias-member will refer to that type.

* + If the global namespace contains a type named B that has N type parameters, the qualified-alias-member will refer to that type constructed with the given type arguments.
  + If the qualified-alias-member is undefined, this will result in a compile-time error.

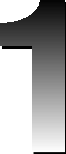
Beginning with the namespace declaration immediately containing the qualified-alias-member (if any), continuing with each enclosing namespace declaration (if any), and ending with the compilation unit containing the qualified-alias-member, the following steps are followed until an entity is found:

* If the namespace declaration or compilation unit contains a using-alias-directive that associates A with a type, the qualified-alias-member is undefined. This will cause a compile-time error.
* Alternatively, if the namespace declaration or compilation unit contains an extern-alias- directive or using-alias-directive that associates A with a namespace, the following set of rules is followed:
  + If the namespace associated with A contains a namespace named B and N is zero, this means that the qualified-alias-member refers to that namespace.
  + If the namespace associated with A contains a nongeneric type named B and N is zero, this means that the qualified-alias-member refers to that type.
  + If the namespace associated with A contains a type named B that has N type parameters, the qualified-alias-member refers to that type constructed with the given type argu- ments.
  + Otherwise, the qualified-alias-member is undefined, which will cause a compile-time error.
* If, after all this, the qualified-alias-member remains undefined, a compile-time error is generated.

##### Summary

In this chapter you looked at namespaces in C# and how they allow the programmer to both organize classes in the .NET Framework and control the scope of class and method names used. By being able to organize both internally and externally, the programmer is able not only to write less code to do the same amount of work but also to write code that’s easier to follow (and later debug). Namespaces also help reduce the risk of naming conflicts with other libraries.

In Chapter 12, we will be looking at classes.

### Classes

In this chapter you look at one of the most important concepts of C# — the class. We’ll begin by looking at what a class is and then declaring classes. Then we will take a closer look at specific aspects of classes.

##### What are Classes?

A class is a programming data structure. Classes can contain the following:

* + Data members (constants and fields)
  + Nested types
  + Function members (events, finalizers, indexers, instance constructors, methods, proper- ties, and static constructors)

All class types support inheritance.

##### Class Declarations

A class declaration is a type of declaration used to declare new classes:

class-declaration: attributesopt

class-modifiersopt

partialopt

class identifier type-parameter-listopt class-baseopt

type-parameter-constraints-clausesopt

class-body

;opt

Class declarations are made up of:

* An optional set of attributes, followed by
* An optional set of class modifiers, followed by
* An optional partial modifier, followed by
* The keyword class and an identifier that assigns a name to the class, followed by
* An optional type-parameter-list, followed by
* An optional class-base specification, followed by
* An optional type-parameter-constraints-clauses, followed by
* A class-body, followed by
* An optional semicolon

If a type-parameter-constraints-clauses is supplied, a type-parameter-list has to also be supplied.

Class Modifiers

Class declarations can contain a sequence of class modifiers:

class-modifiers: class-modifier

class-modifiers class-modifier

class-modifier: new

public protected internal private abstract sealed static

The new modifier is used to specify that a class hides an inherited member of the same name. A compiler error is generated if a new modifier appears on a class declaration that is not a nested class declaration.

The following modifiers control the accessibility of the class:

* Internal — Access limited to the assembly that defines the class
* Protected — Access limited to the containing class or types derived from the containing class
* Private — Access limited to the containing type
* Public — Access not limited
* Abstract — Used to indicate that the class is not complete and that it should only be used as a base class
  + Sealed — Used to prevent derivation from the class (cannot be abstract)
  + Static — Cannot be sealed or abstract, cannot include a base-class specification, cannot contain operators, cannot have members that have protected or protected internal accessibility, and cannot contain static members

A compiler error will be generated if the same modifier is used more than once in a class declaration.

Class Base Specification

A class declaration can include a class-base specification. This is used to define the direct base class of the class and the interfaces implemented by the class:

class-base:

: class-type

: interface-type-list

: class-type , interface-type-list

interface-type-list: interface-type

interface-type-list , interface-type

Base Classes

When a class-type is included in the class-base, it is used to specify the direct base class of the class being declared.

If a nonpartial class declaration doesn’t have a class-base, or if the class-base lists only interface types, the direct base class is an object.

When a partial class declaration includes a base-class specification, that base class will reference the same type as all other parts of that partial type that include a base-class specification.

If no part of a partial class includes a base-class specification, the base class is object.

Interface Implementations

A class-base specification can include a list of interface types. In this case, the class implements the given interface types.

Class Body

The class-body of a class is used to define the members of the class:

class-body:

{ class-member-declarationsopt }

Partial Declarations

The partial modifier is used when defining a class, struct, or interface type in multiple parts. Note, though, that partial is not a keyword.

partial has to appear immediately before one of the keywords class, struct, or interface.

Each part of a partial type declaration has to include a partial modifier and has to be declared in the same namespace or containing type as the other parts. The partial modifier is used to show that the remaining parts of the type declaration might appear elsewhere in the code, although there might not be any additional code.

##### Class Members

The members of a class are made up of the members introduced by its class-member-declarations

and any members inherited from the direct base class.

class-member-declarations: class-member-declaration

class-member-declarations class-member-declaration

class-member-declaration: constant-declaration field-declaration method-declaration property-declaration event-declaration indexer-declaration operator-declaration constructor-declaration finalizer-declaration

static-constructor-declaration type-declaration

Members of a class fall into the following categories:

* **Constants.** Constant values associated with the class
* **Events.** Define notifications that may be generated by the class
* **Fields.** Class variables
* **Finalizers.** Implement the actions performed before class instances are no longer needed
* **Indexers.** Allow instances of the class to be indexed like arrays
* **Instance constructors.** Implement the actions required to initialize the instances of the class
* **Generic and nongeneric methods.** Implement the actions of the class
* **Operators.** Define expression operators applied to the class
* **Properties.** Define the named characteristics and actions performed by the class
  + **Static constructors.** Implement the actions that initialize the class
  + **Types.** Represent the local types of the class

Members that contain executable code are known as the function members of the class. The function members of a class include:

* + Events
  + Finalizers
  + Indexers
  + Instance constructors
  + Methods
  + Operators
  + Properties
  + Static constructors

The following rules apply to class-member-declarations:

* + Instance constructors, finalizers, and static constructors must have the same name as the enclos- ing class.
  + The name of a type parameter in the type-parameter-list of a class declaration has to be dif- ferent from the names of all other type parameters in the same type-parameter-list. It also has to be different from the name of the class and the names of all members of the class.
  + The name of a type has to be different from the names of all nontype members declared in the same class.
  + The names of any constants, fields, properties, or events have to be different from the names of all other members declared in the same class.
  + The name of a method has to be different from the names of all other nonmethods declared in the same class.
  + The signature of an instance constructor has to be different from the signatures of all other instance constructors declared in the same class.
  + The signature of an indexer has to be different from the signatures of all other indexers declared by the class.
  + The signature of an operator has to be different from the signatures of all other operators declared by the class.

Inheritance

A class will inherit the members of its direct base class. The upshot of inheritance is that a class will implicitly contain all members of its direct base class, except for any instance constructors, finalizers, and static constructors.

A derived class can add new members to those it inherits, but it cannot remove the definition of an inherited member.

Instance constructors, finalizers, and static constructors are not inherited, but all other members are.

A class can declare virtual methods, properties, indexers, and events, and derived classes can override the implementation of these function members.

Members inherited from a constructed generic type are inherited after type substitution.

new Modifier

If a new modifier is used in a declaration that doesn’t hide available inherited members, a warning is generated by the compiler.

Access Modifiers

Five access modifiers can be used on class-member-declarations:

* internal
* private
* protected
* protected internal
* public

Apart from protected internal, only one modifier can be used at a given time.

Static/Instance Members

Members of a class are either static members or instance members.

When one of the following declarations includes a static modifier, it declares a static member:

* Constructor
* Event
* Field
* Method
* Operator
* Property

When one of the following declarations does not include a static modifier, it declares an instance member:

* + Constructor
  + Event
  + Field
  + Finalizer
  + Indexer
  + Method
  + Property

Constants

A constant is a class member used to represent a constant value that will be used during compilation:

constant-declaration:

attributesopt constant-modifiersopt const type constant-declarators ;

constant-modifiers: constant-modifier

constant-modifiers constant-modifier

constant-modifier: new

public protected internal private

constant-declarators: constant-declarator

constant-declarators , constant-declarator

constant-declarator:

identifier = constant-expression

The type specified in a constant declaration can be one of the following:

* + bool
  + byte
  + char
  + decimal
  + double
* enum type
* float
* int
* long
* reference type
* sbyte
* short
* string
* uint
* ulong
* ushort

Each constant expression will yield a value that is the same as the target type or a type that can be con- verted to the target type through implicit conversion.

##### Fields

A field is a member used to represent a variable associated with an object or class:

field-declaration:

attributesopt field-modifiersopt type variable-declarators ;

field-modifiers: field-modifier

field-modifiers field-modifier

field-modifier: new

public protected internal private static readonly volatile

variable-declarators: variable-declarator

variable-declarators , variable-declarator

variable-declarator: identifier

identifier = variable-initializer

variable-initializer: expression

array-initializer

Static and Instance Fields

When a field declaration includes a static modifier, the fields will be static. When no static modifier is present, the fields are instance.

Static fields and instance fields are two of the several kinds of variables supported by C# and are referred to as static variables and instance variables.

readonly Fields

When a field declaration makes use of a readonly modifier, the fields introduced by the declaration are read-only.

Any attempt to assign to a readonly field or pass it as an out or ref parameter, other than as a variable declarator or as part of an instance constructor, will result in a compiler error.

Volatile Fields

Volatile fields are declarations that make use of the volatile modifiers. For volatile fields, the opti- mizations performed by the compiler on standard nonvolatile fields are limited to volatile read and volatile writes.

Volatile fields are limited to the following types:

* + Enum type that has one of the following base types:
    - byte
    - int
    - sbyte
    - short
    - uint
    - ushort
  + Reference types
  + Type parameters
  + One of the following types:
    - bool
    - byte
    - char
* float
* int
* sbyte
* short
* uint
* ushort

Field Initialization

The initial value of a field will be the default value of the field’s type, irrespective of whether it is a static field or an instance field.

Variable Initialization

Field declarations can include variable initializers. There are two types:

* **Static fields.** The variable initializers correspond to assignment statements executed during class initialization.
* **Instance fields.** The variable initializers correspond to assignment statements executed when an instance of the class is created.

##### Methods

A method, which is declared using a method declaration, is a member that implements code executed by an object or class:

method-declaration:

method-header method-body

method-header:

attributesopt method-modifiersopt return-type member-name type-parameter-listopt ( formal-parameter-listopt ) type-parameter-constraints-clausesopt

method-modifiers: method-modifier

method-modifiers method-modifier

method-modifier: new

public protected internal private static virtual sealed override

abstract extern

return-type: type void

member-name:

identifier

interface-type . identifier

method-body: block

;

A method-declaration can include:

* + A set of attributes
  + A valid combination of access modifiers:
    - public
    - protected
    - internal
    - private
  + The new modifier
  + The static modifier
  + The virtual modifier
  + The override modifier
  + The sealed modifier
  + The abstract modifier
  + The extern modifier

Method Parameters

The optional parameters of a method are declared by a formal parameter list:

formal-parameter-list: fixed-parameters

fixed-parameters , parameter-array parameter-array

fixed-parameters: fixed-parameter

fixed-parameters , fixed-parameter fixed-parameter:

attributesopt parameter-modifieropt type identifier

parameter-modifier: ref

out

parameter-array:

attributesopt params array-type identifier

The parameter list is made up of one or more comma-separated parameters. Note that only the last parameter can be a parameter array.

A fixed-parameter consists of:

* An optional set of attributes
* An optional ref or out modifier
* A type
* An identifier

There are four kinds of formal parameters:

* **Value parameters.** Declared without any modifiers
* **Reference parameters.** Declared with the ref modifier. A reference parameter does not create a new storage location and must be initialized before passing to a method. Instead, it represents the same storage location as the variable given as the argument in the method invocation.
* **Output parameters.** Declared with the out modifier. An output parameter does not create a new storage location and does not need to be initialized before passing to a method. Instead, it repre- sents the same storage location as the variable given as the argument in the method invocation.
* **Parameter arrays.** Declared with the params modifier. Apart from allowing a variable number of arguments during invocation, a parameter array is equivalent to a value parameter.

Static/Instance Methods

When a method declaration includes a static modifier, that method is static. When there isn’t a static modifier present, the method is an instance.

Virtual Methods

When an instance method declaration includes a virtual modifier, that method is virtual. When no vir- tual modifier is present, the method is nonvirtual.

Override Method

When an instance method declaration includes an override modifier, the method is an override. An override method is used to override an inherited virtual method with the same signature.

A compiler error is generated unless all of the following conditions are true:

* + The overridden base method is virtual, abstract, or override (it cannot be static or nonvirtual).
  + The overridden base method is not sealed.
  + The override declaration and the overridden base method have the same return type.
  + The override declaration and the overridden base method have the same declared accessibility.

Sealed Methods

When an instance method declaration includes a sealed modifier, the method is sealed. A sealed method is used to override an inherited virtual method with the same signature. Using a sealed modifier prevents a derived class from overriding the method.

Abstract Methods

When an instance method declaration makes use of an abstract modifier, that method is abstract.

An abstract method declaration creates a new virtual method but doesn’t provide an implementation of that method. To compensate for this, nonabstract derived classes have to provide their own implementa- tion by overriding that method.

Method Body

The method body of a method declaration is made up of either a block of code or a semicolon.

Since abstract and external method declarations do not provide method implementations, method bod- ies are made up of simply a single semicolon.

For all other methods, the method body is a code block that consists of the statement that needs to be executed when the method is invoked.

##### Properties

A property is a member that allows access to aspects of an object or a class. Properties make use of acces- sors that specify the statements that should be executed when their values are read or written:

property-declaration:

attributesopt property-modifiersopt type member-name { accessor-declarations }

property-modifiers: property-modifier

property-modifiers property-modifier

property-modifier: new

public protected internal private static virtual sealed override abstract extern

Property declarations include:

* A set of attributes
* A valid combination of the access modifiers
  + public
  + protected
  + internal
  + private
* The new modifier
* The static modifier
* The virtual modifier
* The override modifier
* The sealed modifier
* The abstract modifier
* The extern modifier

Static/Instance Properties

When a property declaration uses a static modifier, the property is static. When no static modifier is used, the property is an instance.

Accessors

Accessor declarations of a property specify the statements associated with reading and writing that property:

accessor-declarations:

get-accessor-declaration set-accessor-declarationopt set-accessor-declaration get-accessor-declarationopt

get-accessor-declaration:

attributesopt accessor-modifieropt get accessor-body

set-accessor-declaration:

attributesopt accessor-modifieropt set accessor-body

accessor-modifier: protected internal private

protected internal internal protected

accessor-body: block

;

Accessor declarations are made up of a get-accessor-declaration and/or a set-accessor- declaration. Each accessor declaration is made up of the token get or set, which is followed by an accessor-body.

For abstract and extern properties, the accessor-body for each accessor specified will be nothing more than a semicolon. For the accessors of any nonabstract, nonextern property, the accessor-body is a code block that contains the statements to be executed when the corresponding accessor is invoked.

A get accessor is the same as a parameterless method with a return value of the property type. When a property is referenced in an expression, the get accessor of the property is invoked to work out the value of the property (except where it is the target of an assignment).

Properties are classified as follows:

* + If the property includes both a get accessor and a set accessor, it is a read-write property.
  + If the property has only a get accessor, it is a read-only property.
  + If the property has only a set accessor, it is a write-only property.

Virtual, Sealed, Override, and Abstract Accessors

A virtual property declaration is used to specify that the accessors of the property are virtual.

The virtual modifier will apply to every nonprivate accessor of a property. When an accessor of a

virtual property has the private accessor-modifier, the private accessor is not virtual.

An abstract property declaration is used to specify that the accessors of a property are virtual. However, it doesn’t provide any implementations of the accessors.

A property declaration that has both the abstract and override modifiers is used to specify that the property is abstract and overrides a base property.

Abstract property declarations are only allowed in abstract classes.

The accessors of an inherited virtual property can be overridden in a derived class through the use of a property declaration that uses an override directive, known as an overriding property declaration.

An overriding property declaration can make use of sealed modifiers, which prevent a derived class from further overriding the property.

##### Events

All events are members that enable an object or class to provide notifications. All events are declared using event declarations:

event-declaration:

attributesopt event-modifiersopt event type variable-declarators ; attributesopt event-modifiersopt event type member-name

{ event-accessor-declarations }

event-modifiers: event-modifier

event-modifiers event-modifier

event-modifier: new

public protected internal private static virtual sealed override abstract extern

event-accessor-declarations:

add-accessor-declaration remove-accessor-declaration remove-accessor-declaration add-accessor-declaration

add-accessor-declaration: attributesopt add block

remove-accessor-declaration: attributesopt remove block

An event declaration can include:

* A set of attributes
  + A valid combination of access modifiers:
    - public
    - protected
    - internal
    - private
  + The new modifier
  + The static modifier
  + The virtual modifier
  + The override modifier
  + The sealed modifier
  + The abstract modifier
  + The extern modifier

Field-Like Events

Some events can be used as fields in code (in any location in the code where fields could otherwise be used). Events used as fields cannot be abstract or extern and cannot explicitly include event accessor declarations.

The field will contain a delegate that will refer to the list of event handlers that have been added to the event. If no event handlers have been added, the field contains null.

Static/Instance Events

When an event declaration includes a static modifier, the event is static. When there is no static modifier included, the event is an instance event.

A static event is not in any way linked with a specific instance, and referring to this in an accessor of a static event will result in a compiler error.

Virtual, Sealed, Override, and Abstract Accessors

A virtual event declaration is used to specify that the accessors of that event are virtual. The virtual

modifier will apply to all accessors of an event.

An abstract event declaration is used to specify that any accessors of the event will be virtual, but note that it does not provide any implementation of the accessors. To do this, nonabstract derived classes are needed, which will provide their own implementation for the accessors by overriding the event. Because of this, the accessor body consists of nothing more than a semicolon.

An event declaration that includes both the abstract and override modifiers is used to specify that the event is both abstract and at the same time overrides a base event. Abstract event declarations are only allowed in abstract classes.

Any accessors of an inherited virtual event can be overridden in a derived class when an event declara- tion that specifies an override modifier is used. This technique is known as an overriding event decla- ration. The overriding event declaration is not used to declare a new event; rather, it specializes the implementations of the accessors of an existing virtual event. Any overriding event declaration will have exactly the same accessibility modifiers, type, and name as the overridden event.

It is possible for an overriding event declaration to make use of the sealed modifier, which will prevent a derived class from further overriding the event. The accessors of a sealed event will also be sealed.

##### Indexers

An indexer is a member that allows an object to be indexed in the same way that an array can be indexed. All indexers are declared using an indexer declaration:

indexer-declaration:

attributesopt indexer-modifiersopt indexer-declarator { accessor-declarations }

indexer-modifiers: indexer-modifier

indexer-modifiers indexer-modifier

indexer-modifier: new

public protected internal private virtual sealed override abstract extern

indexer-declarator:

type this [ formal-parameter-list ]

type interface-type . this [ formal-parameter-list ]

An indexer declaration is made up of:

* A set of attributes
* A valid combination of the access modifiers:
  + public
  + protected
    - internal
    - private
* The new modifier
* The virtual modifier
* The override modifier
* The sealed modifier
* The abstract modifier
* The extern modifier

Indexer declarations have to follow the same rules as method declarations regarding the valid combina- tions of modifiers allowed. The only exception is that the static modifier is not permitted on an indexer declaration.

The modifiers virtual, override, and abstract are mutually exclusive, except where the abstract and

override modifiers can be used in combination so that an abstract indexer can override a virtual one.

At first glance, indexers and properties might look similar. There are, however, a number of differences between the two:

* All properties are identified by name, while indexers are identified by their signature.
* Properties can be static members, while indexers are always instance members.
* Properties are accessed through simple names or member access, while an indexer element is accessed using an element access.
* If an indexer accessor tries to declare a local variable or local constant with the same name as an indexer parameter, a compiler error will be generated.
* A get accessor of a property is equivalent to a method with no parameters, while a get accessor of an indexer is equivalent to a method with the same parameter list as the indexer.
* A set accessor of a property is equivalent to a method with a single parameter named value, while a set accessor of an indexer is equivalent to a method with the same formal parameter list as the indexer, with the addition of a parameter named value.

##### Operators

Operators are members used to define the meaning of an expression operator applied to instances of a class:

operator-declaration:

attributesopt operator-modifiers operator-declarator operator-body

operator-modifiers: operator-modifier

operator-modifiers operator-modifier

operator-modifier: public

static extern

operator-declarator:

unary-operator-declarator binary-operator-declarator conversion-operator-declarator

unary-operator-declarator:

type operator overloadable-unary-operator ( type identifier )

overloadable-unary-operator: one of

+

-

!

~

++

--

true false

binary-operator-declarator:

type operator overloadable-binary-operator ( type identifier , type identifier

)

overloadable-binary-operator: one of

+

-

\*

/

%

&

|

^

<<

>>

==

!=

>

<

>=

<=

conversion-operator-declarator:

implicit operator type ( type identifier ) explicit operator type ( type identifier )

operator-body: block

;

There are three categories of operators:

* Unary
* Binary
* Conversion

The following rules apply to all operator declarations:

* All operator declarations have to include both a public and a static modifier.
* The same modifier cannot appear multiple times in an operator declaration.
* All the parameters of an operator will be value parameters.
* The signature of an operator has to be different from the signatures of all other operators declared in the same class.

Unary Operators

The following unary operators all take a single parameter and are able to return any type:

❑ +

* -
* !
* ~

The following unary operators can take a single parameter and return the same type:

❑ ++

* --

The following unary operators can take a single parameter and return the bool type:

* true
* false

Binary Operators

Binary nonshift operators take two parameters and can return any type.

The following operators take two parameters, but the second parameter must be an int. These can return any type:

* <<
* >>

The signature of a binary operator is made up of the operator token and the types of the parameters.

Conversion Operators

A conversion operator declaration is a user-defined conversion operator used to augment the predefined implicit and explicit conversions.

A conversion operator declaration that makes use of the implicit keyword creates a user-defined implicit conversion operator.

A conversion operator declaration that makes use of the explicit keyword creates a user-defined explicit conversion operator.

##### Instance Constructors

Instance constructors are members that implement the actions required to initialize an instance of a class:

constructor-declaration:

attributesopt constructor-modifiersopt constructor-declarator constructor-body

constructor-modifiers: constructor-modifier

constructor-modifiers constructor-modifier

constructor-modifier: public

protected internal private extern

constructor-declarator:

identifier ( formal-parameter-listopt ) constructor-initializeropt

constructor-initializer:

: base ( argument-listopt )

: this ( argument-listopt )

constructor-body: block

;

A constructor declaration can include the following:

* A set of attributes
* A valid combination of the access modifiers:
  + public
  + protected
    - internal
    - private
* An extern modifier

##### Static Constructors

A static constructor is a member that contains the actions needed to initialize a class:

static-constructor-declaration:

attributesopt static-constructor-modifiers identifier ( ) static-constructor-body

static-constructor-modifiers: externopt static

static externopt

static-constructor-body: block

;

A static constructor declaration includes both a set of attributes and an extern modifier.

When a static constructor declaration contains an extern modifier, the static constructor is called an external static constructor. Since external static constructor declarations have no implementation, the body of a static constructor consists of just a semicolon.

##### Finalizers

A finalizer is a member that implements all the actions that need to be carried out to finalize an instance of a class:

finalizer-declaration:

attributesopt externopt ~ identifier ( ) finalizer-body

finalizer-body: block

;

Because finalizers are automatically invoked, they cannot be invoked explicitly. An instance becomes open for finalization at the point where it is no longer possible for any code to use that instance. After that point, the finalizer can be executed at any time after the instance becomes eligible for finalization. This cannot be controlled in code.

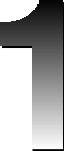
Because a finalizer cannot have any parameters, it cannot be overloaded. This means that a class can have only one finalizer.

*Another word for finalizers is “destructors.”*

##### Summary

In this chapter you looked at classes in C#. A class is a programming data structure and can contain data members, nested types, and functions. All class types support inheritance, and classes form the back- bone of a lot of C# coding, considerably improving modularity.

In Chapter 13, you look at structs.

### Structs

Any C or C++ programmer is likely to have made use of structs. In C++, a struct is very similar to a class, with the exception of the default accessibility of the members. Things are different in C#, and in this chapter you look at the rules for making use of structs in your code.

##### What are Structs?

The word “structs” is short for “structure” — a type of variable. They are called structures because they are constructed of several different pieces of data which may or may not be of the same type. The power of structs comes from the fact that they allow you to define types based on this data structure.

What kind of data lends itself to structs?

* Complex numbers
* Key-value pairs
* Points in a coordinate system (direction and distance travelled)

*Structs are particularly suited to small data structures. Microsoft recommends keeping the size of structs under 16 bytes. Trying to scale them up leads to a lot of extra overhead. The key to data structures is:*

* They have few data members.
* They do not need to use inheritance or referential identity.
* They can be implemented using value semantics where assignment copies the values instead of the reference.

So, why does Java, which is similar to C# in a number of ways, not have structs? The main reason is that it is has the ability to create types with value semantics. These can lead to better perfor- mance in a managed environment (if used properly).

.NET supports the concept of value types and reference types, whereas in Java you have only reference types. All instances of references are allocated to the managed heap and are cleaned up by garbage col- lection when there are no longer references to them. Value types are not allocated to the managed heap but instead are allocated in the stack, and the allocated memory is recovered when scope ends. In C#, all value types are passed by value, while all reference types are passed by reference (pretty obvious, really). All primitive data types in C# apart from System.String are value types.

In C#, structs are always value types, while classes are reference types. Values in C# can be created in one of two ways:

* Using the enum keyword
* Using the struct keyword

The benefit of using a value type instead of a reference type is that it results in fewer objects to manage in the heap, which means less work for garbage collection.

Structs aren’t the solution to all situations, though. Passing a big struct is slower and harder on the sys- tem than passing a corresponding reference. There is also additional overhead when it comes to boxing and unboxing.

The simple types provided by C# (such as int and bool) are all struct types, and it is possible to use struct and operator overloading to implement new primitive types.

##### Struct Declarations

A struct-declaration is a type-declaration that declares a new struct:

struct-declaration: attributesopt

struct-modifiersopt

partialopt

struct identifier

type-parameter-listopt struct-interfacesopt

type-parameter-constraints-clausesopt

struct-body ;opt

A struct-declaration consists of:

* An optional set of attributes, followed by
* An optional set of structmodifiers, followed by
* An optional partial modifier, followed by
* The keyword struct and an identifier that names the struct, followed by
* An optional type-parameter-list, followed by
* An optional struct-interfaces specification, followed by
* An optional type-parameter-constraints-clauses, followed by
  + A struct-body
  + Optionally followed by a semicolon

If a struct declaration supplies a type-parameter-constraint-clause, it must also supply a

type-parameter-list. If a type-parameter-list is supplied in a struct, this is known as a generic struct declaration.

Struct Modifiers

A struct-declaration can contain a sequence of struct modifiers. These are optional. Here is the syntax:

struct-modifiers: struct-modifier

struct-modifiers struct-modifier struct-modifier:

new public protected internal private

Using the same modifier multiple times in the struct declaration will cause a compile-time error. Struct declaration modifiers have the same meaning as those found in class declarations.

Struct Interfaces

A struct declaration can also contain a struct-interface specification. When used, the struct will implement a specific interface type:

struct-interfaces:

: interface-type-list

Struct Body

The struct body is used to define the members that make up the struct.

struct-body:

{ struct-member-declarationsopt }

Struct Members

Struct members consist of:

* + Members added using the struct-member-declarations
  + Members inherited from System.ValueType

The syntax is shown below:

struct-member-declarations: struct-member-declaration

struct-member-declarations struct-member-declaration struct-member-declaration:

constant-declaration field-declaration method-declaration property-declaration event-declaration indexer-declaration operator-declaration constructor-declaration

static-constructor-declaration type-declaration

All of the class-member-declarations are struct-member-declarations, with the exception of

finalizer-declarations.

##### Differences Between Class and Struct

Here is a struct definition in C# code:

public struct Foo

{

private string fooString; private int fooNumber;

public string FooString

{

get

{

}

set

{

}

}

return fooString;

fooString = value;

public int GetFooNumber()

{

return fooNumber;

}

}

This looks very similar to a class. There are, however, a number of key differences between structs and classes. These differences are discussed in the following sections.

Value Semantics

The following are the key differences between structs and classes:

* + Structs are value types (a value type is either a struct type or an enumeration type) and have value semantics.

Struct type variables directly contain the data of the struct.

* + Classes are reference types (a class type, an interface type, an array type, or a delegate type) and have reference semantics.

Class type variables contain only a reference to the data (which is known as an object).

This leads to a subtle difference in the way that structs and classes work. With a struct, each variable has an independent copy of the data, and operations working on one of copy of the data cannot affect other copies.

With classes this is not the case, and operations on one variable affect the object referenced by other vari- ables. This is a key feature, and how you want the code to work will dictate your choice.

*Because structs are not reference types, they cannot have a value of* null*.*

Inheritance

All struct types implicitly inherit from System.ValueType, while classes derive from System.Object or a descendant. It is true that System.ValueType derives from System.Object, but this does not mat- ter, since:

* + Structs cannot derive from any other class or struct.
  + They cannot specify a base class.

Remember, though, that a struct can implement a number of interfaces, and when a struct is treated as an interface, it is implicitly boxed.

Structs cannot be abstract and are always sealed. This means that the following modifiers are not allowed in struct declarations:

* + abstract
  + sealed

Also, since inheritance is not allowed, the declared accessibility of a struct member cannot be set to:

* + protected
  + protected internal

Finally, function members cannot be:

* + abstract
  + virtual

The override modifier can only be used to modify methods inherited from System.ValueType.

Assignments

As mentioned earlier, when assigning to a struct type variable, a copy of the value being assigned is cre- ated. This is a fundamental difference between structs and classes.

When a struct is passed as a value parameter or is returned from a function member, a copy is created that preserves the integrity of the original value.

Structs are passed by reference to functions using the following parameters:

* ref
* out

Default Values

Several kinds of variables are automatically initialized to their default values:

* Static variables
* Instance variables of class instances
* Array elements

The default value depends on the type of variable:

* For a variable of a value-type, the default value is the same as the value computed by the

value-type’s default constructor.

* If the variable is of a reference-type, the default value is null.

However, since structs are a value-type that cannot be set to null, the default value of a struct is the value generated by setting all value type fields to their default value and all reference type fields to null.

Boxing/Unboxing

When a value of a struct is converted to an object type or an interface type implemented by the struct, a boxing operation is carried out. Similarly, when a value of an object or interface type is converted back to a struct type, an unboxing operation is carried out. This boxing or unboxing operation is responsible for copying the struct value into or out of the boxed instance.

this

*This means that changes made to the unboxed struct are not made to the boxed one.*

It is important to understand the meaning of this in regard to structs.

Within the instance construct of a struct, this is equivalent to the out parameter of the struct type. Within an instance function member of a struct, this is equivalent to the ref parameter of the struct type.

In either case, this is still classified as a variable, and the entire struct can be modified by passing this

as a ref or out parameter or assigning to this.

Field Initializers

The default value of a struct consists of the value that is generated by setting all value type fields to their default value and all reference type fields to null. This is the reason why a struct does not allow instance field declarations to include variable initializers.

Constructors

Although allowed by the CLR, C# itself does not allow structs to have a default parameterless construc- tor. The reason for this is that, for a value type, compilers by default don’t generate a default constructor and don’t generate a call to the default constructor. So, even if you define a default constructor, it will never be called.

To avoid such problems, the C# compiler prevents definition of a default constructor by the programmer. Because this default constructor is not generated, fields cannot be initialized when defining them, mean- ing that the following is not allowed:

struct MyFoo

{

int x = 1;

}

Finalizers

Finalizers cannot be declared by a struct.

Static Constructors

Static constructors for structs follow rules very similar to those for classes. Executing static constructors for a struct is carried out by the first occurrence of the following events:

* + An instance member of a struct is referenced.
  + A static member of the struct is referenced.
  + An explicitly declared constructor of a struct is called.

##### When to Use Structs

The key to using structs is to know when to use them and when not to use them. Here’s where structs work great:

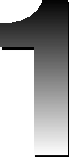
* + You want your type to have the look and feel of a primitive type.
  + You create a lot of instances, use them for a short time, and then get rid of them (say, within a loop).
  + The instances you create are not passed much.
* You don’t want to derive from other types or let others derive from your type.
* You want to operate on a copy of your data. Here’s when not to use structs:
* The size of the struct gets large (that is, the cumulative size of the members). Microsoft recom- mends that you keep this under 16 bytes.
* The operations carried out involve a lot of boxing and unboxing.

##### Summary

In this chapter you looked at structs and how to use them in C#. You saw the type of data best suited to structs and how to declare structs in code before going on to look at struct modifiers, interfaces, bodies, and members.

Then you looked at the key differences between classes and structs before looking at when (and when not) to use structs.

In Chapter 14, you look at how to leverage arrays in C#.

### Arrays

In this chapter you examine arrays and how you can use them in your C# programs. We’ll begin by looking at what arrays are before looking at creating arrays and how to use them in your code.

##### What is an Array?

An array is a data structure commonly used in programming. It is used to hold a number of vari- ables accessed through an index. This index is a number that corresponds to the position of the data within the array (the diagrams that follow will make this clear).

Arrays are classified based on their rank. The rank determines the number of indices associated with a particular array. The rank of an array is also known as a dimension, and this is also used when referring to an array.

An array that has a rank of one is called a single-dimensional array, while any array with a rank greater than one is called a multidimensional array. Multidimensional arrays of a specific size can be referred to more specifically. For example, an array with two ranks is often called a two-dimensional array, while an array with a rank of three is called a three-dimensional array. The diagrams in Figure 14-1 and 14-2 describe arrays in a visual way.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |

Figure 14-1

Figure 14-2

Along with a rank, an array has a length. In fact, each dimension of an array has an associated length. The length of any dimension of an array is always an integer number greater than or equal to zero. It is important to note that these dimension lengths do not form part of the type of any array. Instead, they are determined when the array is created at runtime. This length determines the valid range of indices for that dimension. For a dimension of an array with length N, the indices can range from 0 to N, includ- ing 1. See Figure 14-3 for clarification.

2

1

1

2

1 2 3 4

Figure 14-3

*Zero-dimension arrays are not supported.*

The total number of elements (or vectors) that an array holds is determined by the product of the lengths of each dimension of the array. For example, if you have a three-dimensional array, with each dimension having a length of 4, the total number of elements that the array holds is 64 (444). All this information is included in any signature of the array type and can be marked as statically supplied (that is, fixed) or dynamically supplied (see Figure 14-4).

If one or more of the array dimensions have a zero length, the array is said to be empty.

The element type of an array can be any type, including an array type. Exact array types are created automatically at runtime as required, and no separate definition is required.

*An array of any given type can only hold elements of that type.*

x4

x4

x4

Figure 14-4

##### Array Types

An array type is written as a nonarray type (that is, any type that is not an array type) followed by one or more rank specifiers:

array-type:

non-array-type rank-specifiers

non-array-type: value-type class-type interface-type delegate-type type-parameter

rank-specifiers: rank-specifier

rank-specifiers rank-specifier

rank-specifier:

[ dim-separatorsopt ]

dim-separators:

,

dim-separators ,

The rank of any array type is determined by the leftmost rank specifier in the array type. A rank specifier indicates that an array has a rank of the number of “,” tokens in the rank specifier plus one:

int[] SingleDimensional; SingleDimensional = new int[12];

int[,] TwoDimensional; TwoDimensional = new int[12,24];

int[,,] ThreeDimensional; ThreeDimensional = new int[12,24, 36];

An element type of any array type is the type resulting from deleting the rank specifier on the left:

* T[R] — An array with rank R and a nonarray type T
* T[R][R1]...[RN] — An array with the rank R and an element type T[R1]...[RN]

All rank specifiers are read from left to right before the final nonarray element type. For example:

int[,][,,][]

The type in this example is a two-dimensional array of three-dimensional arrays of single-dimensional arrays of int, while the following

int[,,][][,]

is a three-dimensional array of single-dimensional arrays of two dimensional arrays. The value of an array at runtime can be one of the following:

* null
* A reference to an instance of an array type
* A reference to an instance of a covariant array type

System.Array Type

System.Array is not an array type; it is a class type from which all array types are derived.

System.Array is an abstract type and cannot be instantiated.

The System.Array type is the abstract type base for all array types used in C#. There is an implicit ref- erence conversion from any array type to System.Array and from any interface type implemented by the System.Array type to any array type.

The runtime value of System.Array can be either:

* null
* A reference to an instance of an array type

##### Creating Arrays

Instances of arrays are explicitly created using array-creation expressions or by field or local variable declarations that include array initializers. Arrays can also be generated implicitly using a method.

After an array is created, the rank and length of all dimensions are fixed and cannot change for the life of the instance. Changes to the rank or length dimensions of any current array are not permitted.

An instance of an array will always be an array type. The elements of any array created using an expres- sion will always be initialized to their default values. (In other words, variables of a value type have a default value the same as the value determined by the value type’s default constructor, while reference type variables have a default value of null.)

Accessing Array Elements

You can access array elements by making use of element-access expressions that follow the form:

A[I1, I2,... IN]

Where A is an array-type expression and each instance of Ix is an expression of the following types:

* + int
  + uint
  + long
  + ulong
  + Also any type that can be implicitly converted to one or more of the preceding types

The outcome of accessing any array element is a variable that will itself have the value of the array ele- ment selected.

The elements of an array are enumerated using a foreach statement:

int[] numbers = {1, 2, 3, 4, 5, 6}; foreach (int i in numbers)

{

System.Console.WriteLine(i);

}

Array Members

Each array type inherits the members declared by the System.Array type.

Array Covariance

Array covariance can be somewhat difficult to grasp. Let’s take two reference types, A and B. If an explicit or implicit reference conversion exists from A to B, the same reference conversion also exists

from A[R] to B[R], where R is a rank specifier. That is array covariance. Array covariance means that a value of an array of type A[R] can be a reference to an instance of array type B[R] if an implicit refer- ence exists from B to A.

Because array covariance exists, assignments to array elements incorporate a runtime check to make sure that the value being assigned to the array element is valid (either null or an instance of a type compatible with the element type of array).

class Test

{

static void Fill(object[] array, int index, int count, object value) { for (int i = index; i < index + count; i++) array[i] = value;

}

static void Main() {

string[] strings = new string[100]; Fill(strings, 0, 100, “Undefined”);

Fill(strings, 0, 10, null);

Fill(strings, 90, 10, 0); // Fail System.ArrayTypeMismatchException thrown

}

}

In the preceding examples, the assignment to array[i] and Fill methods include the runtime check.

Array Initializers

Array initializers can be specified in the following locations within C# code:

* Field declarations
* Local variable declarations
* Array creating expressions

The syntax of array initializers is shown as follows:

array-initializer:

{ variable-initializer-listopt }

{ variable-initializer-list , }

variable-initializer-list: variable-initializer

variable-initializer-list , variable-initializer

variable-initializer: expression

array-initializer

Array initializers are made up of a sequence of variable initializers. The variable initializers are enclosed by { and } tokens and separated using , tokens. The variable initializers are themselves expressions, except in the case of multidimensional arrays, where they are nested array initializers.

The context of array initializers is used to determine the type of the array initialized. The array type immediately precedes the initializer in array-creating expressions, while in field or variable declarations, the array type is the field or variable being declared.

Array initializers used in field or variable declarations are purely a shorthand form of an array-creating expression.

For example, this:

int[] a = {5, 4, 3, 2, 1};

Is shorthand for the following:

int[] a = new int[] {5, 4, 3, 2, 1};

For single-dimensional arrays, the array initializer consists of a sequence of expressions compatible with the type of the elements contained in the array. Expressions initialize the array elements in increasing order, starting with the element at the index zero. The number of expressions used in the array initializer gives the length of the array being created. The following example creates an int[] instance that has the length 5.

a[0] = 5; a[1] = 4; a[2] = 3; a[3] = 2; a[4] = 1;

This expression also initializes the instance with the values specified.

When dealing with multidimensional arrays, the array initializer has levels of nesting equivalent to the number of dimensions in the array. The outermost nesting level corresponds to the leftmost array dimen- sion, while the innermost nesting level corresponds to rightmost array dimensions. The length of the dimen- sions of the array is controlled by the number of elements at the appropriate nesting level in the initializer.

Take a look at the following example:

int[,] b = {{8, 9}, {6, 7}, {4, 5}, {2, 3}, {0, 1}};

The preceding expression creates a two-dimensional array with a length of 5 for the leftmost dimension and a length of 2 for the rightmost dimension (a 5 by 2 array):

int[,] b = new int[5, 2];

The expression also initializes the array with the following values:

b[0, 0] = 8;

b[0, 1] = 9;

b[1, 0] = 6;

b[1, 1] = 7;

b[2, 0] = 4;

b[2, 1] = 5;

b[3, 0] = 2;

b[3, 1] = 3;

b[4, 0] = 0;

b[4, 1] = 1;

If an array creating expression contains both explicit dimension lengths and an array initializer, then the lengths will be a constant expressions and the number of elements at the nesting levels will have to match the appropriate nesting length.const int i = 5;

int[] x = new int[5] {1, 2, 3, 4, 5};

int[] y = new int[i] {1, 2, 3, 4, 5};

This line of code does not compile, because the number of initializers exceeds the dimension length:

int[] z = new int[5] {1, 2, 3, 4, 5, 6};

The same is true of the following lines of code:

int i = 2;

int[] x = new int[5] {1, 2};

int[] y = new int[i] {1, 2};

int[] z = new int[5] {1, 2, 3, };

These two lines of code are valid:

int[] x = new int[5] {1, 2, 3, 4, 5};

int[] x = new int[2] {1, 2};

These two lines of code generate a compiler error because the dimension length expression is not a constant:

int[] y = new int[i] {1, 2, 3, 4, 5};

int[] y = new int[i] {1, 2};

These two lines of code generate a compiler error because there is a discrepancy between the number of elements used and the length specified.

int[] z = new int[5] {1, 2, 3, 4, 5, 6};

int[] z = new int[5] {1, 2, 3, };

Trailing Commas

Note that just like C++, C# allows you to have trailing commas present at the end of an array initializer in your source code.

For example, both of the following are valid examples of array initializers:

int[] x = new int[5] {1, 2, 3, 4, 5,};

int[] x = new int[2] {1, 2,};

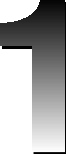
This provides you with far greater flexibility when you are adding or deleting members. You can, for simplicity, add members and their respective commas in pairs. This is particularly useful when you want to write code that will generate such lists or array members automatically.

##### Summary

In this chapter you looked at how to create arrays in C#.

You examined what arrays are, arrays of different dimensions, and the array types that can be used. You also examined array elements (also known as vectors) and looked at how they can be accessed using the foreach statement, before moving on to look at array members, array covariance, and array initializers.

In Chapter 15, we will look at interfaces and how to use them in C#.

### Interfaces

In this chapter you look at a misunderstood and often neglected aspect of C# programming — interfaces. Knowing how to make use of interfaces can allow you to create components that can be interchanged easily.

##### What is an Interface?

The C# specification defines an interface as a defined contract. In addition, structs of classes that implement the interface have to adhere to the contract. This is a somewhat vague description of an interface.

In code, an interface looks very much like a class. The main difference is that it doesn’t have any implementations. The only things that an implementation contains are definitions of events, index- ers, methods, and/or properties.

Why do interfaces provide only the definitions? They are inherited by classes and structs, which provide the implementation for derived interface members.

So, why use interfaces? The main benefit of using interfaces is that programmers can create situa- tions where components in a program can be interchangeable. These will all implement the same interface, so no extra coding is needed to make this work. By using interfaces, the component will expose only certain public members that can be made use of.

Because interfaces must be defined by inheriting classes and structs, they define a contract. But what does this contract stuff mean? For instance, if class ExClass inherits from the IDisposable interface, it is making a contract where it guarantees it has the Dispose() method (which is the only member of the IDisposable interface). Any code that wishes to use class ExClass can check to see if class ExClass inherits IDisposable. When the answer is true, the code knows that it can call ExClass.Dispose().

##### Defining an Interface

Here’s how an interface is defined in code:

interface IExampleInterface

{

void InterfaceMethods();

}

This code defines an interface called IExampleInterface.

*Note that it is common practice to prefix interface names with* I*.*

This interface contains a single method: InterfaceMethods(). However, note here that the method doesn’t have any implementations (no code between curly braces), and also note that it ends with a semicolon.

Here’s how that interface could be implemented:

class Implementer : IExampleInterface

{

static void Main()

{

Implementer iImpInt = new Implementer(); iImpInt.InterfaceMethods();

}

public void InterfaceMethods()

{

Console.WriteLine(“Hello, World!”);

}

}

Let’s now take a closer look at the rules and syntax of using interfaces.

##### Interface Declarations

Interface declarations are type declarations that declare new interface types:

interface-declaration:

attributesopt interface-modifiersopt partialopt interface identifier type- parameter-listopt

interface-baseopt type-parameter-constraints-clausesopt interface-body ;opt

An interface-declaration consists of:

* An optional set of attributes, followed by
* An optional set of interface-modifiers, followed by
  + An optional partial modifier, followed by
  + The keyword interface and an identifier that names the interface, followed by
  + An optional type-parameter-list, followed by
  + An optional interface-base specification, followed by
  + An optional typeparameter-constraints-clauses, followed by
  + An interface-body, optionally followed by
  + A semicolon

An interface declaration cannot supply a type-parameter-constraints-clauses unless it also supplies a type-parameter-list.

An interface declaration that provides a type-parameter-list is generic.

Modifiers

An interface-declaration can optionally include a sequence of interface modifiers:

interface-modifiers: interface-modifier

interface-modifiers interface-modifier

interface-modifier: new

public internal protected private

You cannot have the same modifier appear multiple times in an interface declaration without generating a compiler error. Also, the new modifier is permitted only on nested interfaces.

Four modifiers (public, protected, internal, and private) are used to control accessibility to the interface.

Explicit Base Interfaces

An interface can inherit from one or more other interfaces. These are called the explicit base interfaces of the interface that inherits from them.

When an interface has one or more explicit base interfaces, the interface identifier has a colon added at the end and a comma-separated list of the base interfaces in the interface declaration:

interface-base:

: interface-type-list

Interface Body

The interface body is used to define the members of an interface:

interface-body:

{ interface-member-declarationsopt }

Interface Members

The members of an interface consist of the members inherited from the base interfaces and the members declared by the interface itself (an interface declaration can validly consist of zero members):

interface-member-declarations: interface-member-declaration

interface-member-declarations interface-member-declaration

interface-member-declaration: interface-method-declaration interface-property-declaration interface-event-declaration interface-indexer-declaration

All interface members implicitly have public access, and it will result in a compiler error if interface member declarations include any modifiers.

Interface Methods

Interface methods are declared using interface-method-declarations:

interface-method-declaration:

attributesopt newopt return-type identifier type-parameter-listopt

( formal-parameter-listopt ) type-parameter-constraints-clausesopt ;

The following all have the same meaning as for a method declaration in a class (which is not surprising, given that interfaces are almost identical to classes):

* attributes
* return-type
* identifier
* formal-parameter-list

Interface Properties

Interface properties are declared using interface-property-declarations:

interface-property-declaration:

attributesopt newopt type identifier { interface-accessors }

interface-accessors: attributesopt get ; attributesopt set ;

attributesopt get ; attributesopt set ;

attributesopt set ; attributesopt get ;

The following all have the same meaning as for property declarations in a class:

* + attributes
  + type
  + interface

Interface Events

Interface events are declared using interface-event-declarations:

interface-event-declaration:

attributesopt newopt event type identifier ;

The following all have the same meaning as for event declarations in a class:

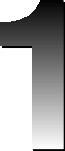
* + attributes
  + type
  + interface

##### Summary

This chapter has taken a brief look at interfaces in C#.

The chapter started by looking at what interfaces are and how they help the programmer write code that is easier to compartmentalize and replace. You also looked at the differences among interfaces, classes, and structs.

Finally, you looked at defining interfaces and also at explicit base interfaces and how they work. In Chapter 16, you look at enums.

### Enums

In this chapter we are going to examine enums, which are strongly typed constants. They are unique types that allow the programmer to assign a name of integral values in code. Since enums are strongly typed, this means that an enum of one type can’t be assigned to an enum of another type.

The purpose of enums is to declare a set of constants in the code. Declaring constants is done as follows:

enum Fruit

{

Apple, Orange, Pineapple Banana

}

This declares an enum called Fruit that has four members:

* + Apple
  + Orange
  + Pineapple
  + Banana

Here is another example of enums in action. In the following code, we have a switch statement controlled by the value of the enum:

using System; public enum Lights

{

Red, Green, Blue

}

class EnumSwitch

{

static void Main()

{

Lights myLights = Lights.Green;

switch (myLights)

{

case Lights.Red:

Console.WriteLine(“The light has been changed to red.”); break;

case Lights.Green:

Console.WriteLine(“The light has been changed to green.”); break;

case Lights.Blue:

Console.WriteLine(“The light has been changed to blue.”); break;

}

Console.ReadLine();

}

}

##### Enum Declarations

Enum declarations are used to declare new enum types.

An enum declaration begins with the keyword enum and defines the following:

* Name
* Accessibility
* Underlying type
* Members

The following shows the syntax for using enum: enum-declaration:

attributesopt enum-modifiersopt enum identifier enum-baseopt enum-body ;opt

enum-base:

: integral-type

enum-body:

{ enum-member-declarationsopt }

{ enum-member-declarations , }

Every enum has an integral type called an underlying type. This is used to represent all the enumerator values defined by the enum. Using explicit declaration, the following underlying types can be declared:

* + byte
  + sbyte
  + int
  + uint
  + long
  + ulong
  + short
  + ushort

Declarations that are not explicit will have the underlying type of int.

*The char type cannot be used as an underlying type.*

The following declares an enum with an underlying type of long:

enum Fruit: long

{

Apple, Orange, Banana

}

Note that a trailing comma is allowable in enum declarations, as they are in array initializers:

enum Fruit: long

{

Apple, Orange,

Banana,

}

##### Enum Modifiers

Enum declarations can contain one or more optional enum modifiers:

enum-modifiers: enum-modifier

enum-modifiers enum-modifier

enum-modifier: new public protected internal private

Entering the same modifier more than once into an enum declaration will cause a compile-time error. The following are access modifiers for enum declarations:

* **Public.** Access to the member is not limited.
* **Protected.** Access to the member is limited to the containing class or types derived from the containing class.
* **Internal.** Access is limited to the classes contained in the assembly.
* **Private.** Access is limited to the containing type.

*Neither the* abstract *nor* sealed *modifiers are allowed in an enum type.*

##### Enum Members

The body of an enum type declaration can contain zero, one, or more enum members. These are named constants of the enum type. As such, no two members can have the same name:

enum Fruit: long

{

Apple, Orange, Banana,

Orange

}

enum-member-declarations: enum-member-declaration

enum-member-declarations , enum-member-declaration

enum-member-declaration: attributesopt identifier

attributesopt identifier = constant-expression

Each enum member will have an associated constant value. The type of this value will be the underlying type for the containing enum. The constant value for each enum member must fall in the range of the underlying type for the enum:

enum Fruit: uint

{

Apple = 5,

Orange = -8, Banana = 5

}

It is possible for members to share the same associated values, as shown below:

enum Fruit

{

Apple = 5,

Orange = -8,

Banana = 5,

orangeFruit = Orange yellowFruit = Banana

}

Here orangeFruit and Orange will have the same value, as will yellowFruit and Banana.

The associated value of an enum member can be assigned either implicitly or explicitly. If the declaration of the enum member has a constant-expression initializer, the value of that constant expression is the associated value of the enum member. If the declaration of the enum member doesn’t have an initializer, its associated value is set implicitly using the following rules:

* + If the enum member is the first one declared, the associated value is zero.
  + If it is not the first, the value is the value of the previous enum value increased by one.

##### Beware Circular References

When using enums, one thing to be wary of is creating circular references between members. These aren’t allowed and will result in a compile-time error:

enum Fruit

{

Banana = yellowFruit, yellowFruit

}

In the preceding example, there is an explicit dependency between Banana and yellowFruit and an implicit dependency between yellowFruit and Banana.

##### System.Enum

The abstract base class of all enum types is the type System.Enum (which is a class type rather than an enum type).

Members inherited from this class are available to all enum types. There is a boxing conversion from any enum type to System.Enum and a corresponding unboxing conversion from System.Enum to any other enum type.

Enum Values and Operations

Each and every enum type defines a distinct type, and an explicit enumeration conversion is required to convert between different enum types or between enum types and integral types. The values that an enum type can take on are restricted only by the enum members.

Enum members have the type of their containing enum type. The value of an enum member declared in enum type E with the associated value v is (E)v.

The following operators can be used on enum type values:

❑ ==

* !=
* <=
* >=
* <
* >

❑ +

* -
* ^
* &
* |
* ~

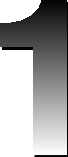
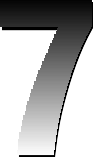
❑ ++

* --
* sizeof

##### Summary

In this chapter you examined enums and how they can be used to declare a set of constants in C# code. Enums are extremely simple to use yet extremely powerful and useful. The main thing to beware of when using them is making a circular reference between members — this is by far the most common error made when using enums.

In Chapter 17, you look at delegates and how to use them in C#.

### Delegates

This is a short chapter on delegates on C#, because they are quite a complex and difficult aspect of C# and are used primarily when dealing with the user interface for Windows Forms. As such, most of this topic is beyond the scope of this book.

Delegates in C# (and in other programming languages such as Java) allow you to do things that other languages do through leveraging function pointers. In C++ there is a feature called a call- back function that uses pointers to functions to pass them as parameters to other functions. The main difference between delegates and function pointers is that delegates are both object-oriented and type-safe, and the delegate encapsulates both the object instance and a method (this encapsu- lation protects data from corruption by other functions because of errors in programming).

A delegate can hold references to one or more functions and invoke them as needed. Delegates differ in other ways from function pointers:

* + Delegates are dynamic and are declared at runtime. In C++ you had to know the function name before you were able to use the function pointer.
  + Delegates don’t just point to one function. Instead, they point to an ordered set of functions.

##### Delegates in Action

A delegate declaration defines a class that is itself derived from the class System.Delegate. As is pointed out in the chapter’s introduction, the delegate instance encapsulates one or more than one method, and each of these is called a callable entity. The contents of a callable entity depend on the type of method:

* + **Instance methods**

Here the callable entity consists of an instance and a method on that instance.

* + **Static methods**

Here the callable entity consists of a method alone.

With an instance of a delegate and an appropriate set of arguments, it is possible to invoke all the instance methods of the delegate.

##### Delegate Declarations

A delegate declaration is a type declaration that allows the declaration of a new delegate type:

delegate-declaration:

attributesopt delegate-modifiersopt delegate return-type identifier type-

parameter-listopt

( formal-parameter-listopt ) type-parameter-constraints-clausesopt ;

delegate-modifiers: delegate-modifier

delegate-modifiers delegate-modifier

delegate-modifier: new

public protected internal private

You should not have multiple instances of the same delegate modifier in a delegate declaration. If you allow this to happen, you will be reminded to correct this oversight by the compile-time error that will be generated.

Note that you can only use the new modifier on delegates that have been declared within another type. When you do this, the delegate will hide all inherited members by the same name.

Modifiers

Four modifiers control the accessibility of the delegate type:

* Public — This declares that access is not limited in any way.
* Private — Here access is limited to the containing type.
* Protected — Here access is limited to the containing class or types derived from the containing class.
* Internal — Access is limited to the classes defined in the same assembly as the delegate.

Depending on the context of the delegate declaration, some of these modifiers might not be allowed. The formal-parameter-list is optional. This specifies the parameters of the delegate, while return-type is used to indicate the delegate’s return type. In other words, the signatures of the functions assigned to the delegates must be identical.

The method and delegate type are consistent if, and only if, the following is true:

* + For each of the parameter methods:
    - If the parameter has no out or ref modifier, the corresponding parameter has no out or ref modifier either. Also, there must exist an identity conversion or implicit reference conversion from the appropriate delegate parameter to the method parameter type.
    - If the parameter does have an out or ref modifier, the corresponding parameter of the delegate type has the same modifier. The corresponding delegate parameter type must be the same as the method parameter type.
  + There must be an implicit reference conversion or identity conversion from the return type of the method to the return type of the delegate.

It is important to remember that delegate types in C# are name equivalent, not structurally equivalent. This means that you can have two delegate types that have the same parameter lists and return types still considered different delegate types.

Declaring Delegates

Delegate types can only be declared using a delegate declaration. All delegate types are derived from the System.Delegate, and they are implicitly sealed. This means that a type cannot be derived from any delegate type. It is also not possible to derive nondelegate class types from System.Delegate. (It is not a delegate type but rather a type class.)

Invocation List

We’ve already mentioned that delegates are used to encapsulate methods. The set of methods encapsu- lated is called an invocation list. If the delegate is created from a single method, the invocation list cre- ates only one entry. When two or more non-null delegate instances are combined, their invocations lists will be concatenated to form a new invocation list. This list will contain two or more entries.

*An invocation list cannot be empty.*

Two invocation lists are concatenated in the order of left operand followed by right operand to form a new invocation list.

Delegates are combined using both binary + and += operators. Delegates can be removed using the binary - and -= operators. Delegates can also be checked for equality.

The following code snippet shows the delegates in action:

delegate void D(int x); class DelEx

{

public static void M1(int i) {...} public static void M2(int i) {...}

}

class Demo

{

static void Main() {

D ex1 = new D(DelEx.M1); D ex2 = new D(DelEx.M2); D ex3 = ex1 + ex2;

D ex4 = ex2 + ex1; D ex5 = ex3 + ex1; D ex6 = ex4 + ex3; D ex7 = ex5 -= ex1;

}

}

The preceding is an example where invocation lists are combined and also where a method is removed. After ex1 and ex2 have been instantiated, each one encapsulates a single method (M1 and M2, respec- tively). When ex3 is then instantiated, it contains two methods in the invocation list (M1 and M2, in that order). Next, ex4 is instantiated, and this again, like ex3, contains two methods, only in a different order (M2 and M1).

When ex5 is instantiated, it now contains three methods (M1, M2, and M1) through combining the invoca- tion lists of ex3 (containing M1 and M2) and ex1 (containing M1). Instantiating ex6 combines the invocation lists of ex4 (M2 and M1) and ex3 (M1 and M2) to encapsulate M2, M1, M1, and M2, respectively.

Instantiating ex7 takes the invocation list of ex5 (M2, M1, M1, and M2) and removes from this the invoca- tion list of ex1 (M1) to leave M2, M1, and M2.

Delegate Instantiation

Instances of delegates are created using a delegate-creation expression or through an implicit conversion from a method group or anonymous method to a delegate type. The delegate then refers to one or more:

* Static methods
* Non-null target objects and instance methods The following shows delegate instantiation in action:

delegate void D(int x);

class DelEx

{

public static void M1(int i) {...} public void M2(int i) {...}

}

class Test

{

static void Main() {

D ex1 = new D(DelEx.M1); Test t = new DelEx();

D ex2 = new D(t.M2);

D ex3 = new D(ex2);

}

}

In the preceding code, the following are created:

* + A static method — D ex1 = new D(DelEx.M1);
  + An instance method — D ex2 = new D(t.M2);
  + A new delegate — D ex3 = new D(ex2);

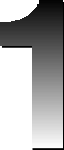
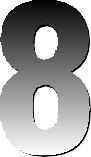
*Once instantiated, an instance of a delegate always refers to the same list of target objects and methods.*

##### Summary

In this chapter you looked at a special feature of C# called delegates and at how these are used to encap- sulate methods to make C# coding both easier and less time consuming.

You looked at how to declare delegates and also how methods are encapsulated into an invocation list. You looked at how to combine invocation lists, as well as at how to remove methods from a list. Finally, you looked at how to instantiate delegates, which is done through a delegate-creation expression or through an implicit conversion from a method group or anonymous method to a delegate type.

In Chapter 18, you look at exceptions and how they are handled in C#.

### Exceptions

Exceptions are a fact of life. Any time you are going to write code, you are going to encounter some mistakes. Even if you write 100-percent, totally error-free code, that doesn’t mean you don’t need to think about exceptions and exception handling — if you write a program that performs some numerical calculations and the user inputs characters that aren’t numbers into the program, the program will run into trouble, and you need to plan for it.

To handle potential problems, C# makes use of exceptions. If you are accustomed to using C++, the exception-handling abilities of C# will be familiar to you. In fact, there are only three important differences:

* + Exceptions in C# are represented by an instance of a class derived from

System.Exception, as opposed to being any value of any type.

* + System-level exceptions such as divide-by-zero have well-defined exception classes.
  + Finally, a block can be used to write code that executes both normally and under conditions of exception.

Exceptions allow the programmer to cater to system-level and application-level errors in C# in a structured, type-safe, and standardized way.

##### Throwing Exceptions

There are two ways that an exception can be thrown:

* + **Using a** throw **statement.** This throws an exception both immediately and uncondition- ally. With a throw statement, control is never passed to the statement that follows the throw.
  + **An exceptional condition arises.** A common example is the divide-by-zero where the system throws a System.DivideByZeroException when the denominator of a division is zero.

##### System.Exception

The base class for all exceptions is the System.Exception class. There are two properties of this class that all exceptions thrown have in common:

* Message — This is a read-only property that contains a human-readable string (of the type

string) that describes the exception.

* InnerException — This is a read-only property of the type Exception. If the value is not null, it refers to the exception that caused the exception. If the value is null, this means that the exception was not caused by another exception.

The specific values of these properties can be specified in calls to the constructor for

System.Exception.

##### Common Exception Classes

The following table shows a list of common exception classes that can be used in C# programs:

**Class Description**

System.ArithmeticException System.ArrayTypeMismatchException

System.DivideByZeroException System.IndexOutOfRangeException System.InvalidCastException

System.NullReferenceException

System.OutOfMemoryException System.OverflowException

System.StackOverflowException

System.TypeInitializationException

Base class for exception for arithmetic oper- ations

Thrown when the type of an element is not compatible with the runtime type of the array

Thrown when a division by zero is carried out

Thrown when trying to index an array that is less than zero or out of bounds

Thrown when an explicit conversion is from a base or interface to a derived class fails (at runtime)

Thrown when a null is used but a refer- enced object is needed

Thrown when memory allocation fails

Thrown when a checked context arithmetic operation overflows

Thrown when an execution stack has too many pending method calls (usually as a result of recursion)

Thrown when a static constructor throws an exception but there is no catch available

Exceptions

##### Handling Exceptions

All exceptions in C# (as with C++) are handled by try statements. The roadmap for handling exceptions is as follows:

* + An exception occurs.
  + The system searches to locate the appropriate catch clause to handle the exception.
    - The current method is searched for a try statement. If found, the catch clauses are processed in order.
    - If the preceding doesn’t yield an appropriate try statement, the method that called the method that threw the exception is examined for a try statement.
    - This process continues until a catch clause that can handle the exception is discovered (an exception that is of the same class, or a base class, of the runtime exception). If a catch clause does not name an exception class, it can handle any exception.
  + The system executes any clauses associated with the try clause.
  + When a matching catch clause is found, the system gets ready to transfer control to the statements in the clause (in order).

What If No Catch Clause Is Found?

At the end of the search outlined above, what if no appropriate catch clause is found?

*Where at all possible, you want to try to avoid having uncaught exceptions, because the behavior of such exceptions will be unspecified.*

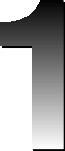
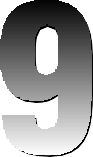
* + If the search for a matching catch clause encounters either a static constructor or static field ini- tializer, a System.TypeInitializationException is thrown. The inner exception of the System.TypeInitializationException will contain the exception that was initially thrown.
  + If the search for a matching catch clause encounters the code that initially began the thread, execution of the thread will be terminated.

##### Summary

In this short chapter you looked at exceptions. The chapter began by looking at some of the major differ- ences between C# exceptions and exceptions in C++. The chapter then covered throwing exceptions, the System.Exception class, common exception classes in C#, and how exceptions are handled.

In Chapter 19, you look at C# attributes.

229

### Attributes

One of the most powerful features of the .NET language is the ability it offers to define custom attributes in the source code (such as methods, classes, and so on). This allows for a concise yet powerful metaprogramming syntax. In this chapter we are going to look at how to use attributes in C# by first introducing you to attributes before looking at a number of different attributes and how to use them in code.

##### Introduction to Attributes

Attributes in C# provide a system for defining declarative tags. These are placed on certain entities in the source code to specify additional information. This information can later be retrieved at run- time using a technique called reflection.

Two kinds of attributes can be used:

* + Predefined attributes
  + Custom attributes

Attributes are defined using attribute classes (covered in the following sections) that can have both positional and named parameters. These attributes are attached to entities using attribute specifi- cations. These can subsequently be retrieved at runtime using attribute instances.

Here is how you declare an attribute in C#:

public class MyNewAttribute : System.Attribute

##### Attribute Classes

Any class that derives directly or indirectly from the abstract class System.Attributes is an attribute class.

The declaration of an attribute class defines a completely new attribute that can be placed in a declaration. It is convention for attribute classes to have the suffix Attribute. In coding, this may or may not be included.

Positional vs. Named Parameters

Attribute classes can have both positional and named parameters:

* **Positional parameters.** Each public instance constructor of an attribute class defines a sequence of positional parameters for that attribute class.
* **Named parameters.** Each nonstatic public read-write field and property of an attribute defines a named parameter of the attribute class.

Attribute Usage

The attribute used to describe how an attribute can be used is AttributeUsage. This attribute has a positional parameter that enables an attribute class to specify the types of declarations that can be used.

The syntax of the code is shown as follows:

using System;

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]

public class ExampleAttribute: Attribute

{

...

}

The preceding sample code defines an attribute class called ExampleAttribute. This can be placed on class declarations and interface declarations.

Here is another example:

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Constructor | AttributeTargets.Field | AttributeTargets.Method | AttributeTargets.Property, AllowMultiple = false)]

AttributeUsage has a named parameter called AllowMultiple. This is used to indicate whether the attribute can be specified more than once for a given entity.

* If AllowMultiple for an attribute class is true, that attribute class is set as a multiuse attribute class and can be specified once or more than once on an entity.
* If AllowMultiple for an attribute class is false or unspecified, that attribute class is set as a single-use attribute class and can be specified no more than once on an entity.

AttributeUsage has another named parameter, called Inherited, used to indicate whether the attribute, when used on a base class, is also inherited to classes derived from that base class.

There are two possible values:

* + If Inherited is set to true, the attribute is inherited.
  + If Inherited is set to false, the attribute is not inherited.

Types of Attribute Parameters

The types of both positional and named parameters for attribute classes are confined to attribute parameter types.

The attribute parameter types are:

* + One of the following types:
    - bool
    - byte
    - char
    - double
    - float
    - int
    - long
    - short
    - string
  + The System.Type type
  + The object type
  + An enum type (as long as it is set to have public accessibility, along with any nested types)
  + A single-dimensional array of any of the preceding

##### Attribute Specification

Attribute specification is where a previously defined attribute is used in a declaration.

Attributes can be specified at the following:

* + global scope
  + type-declarations
  + struct-member-declarations
  + interface-member-declarations
  + class-member-declarations
  + enum-member-declarations
* accessor-declarations
* event-accessor-declarations
* elements of formal-parameter-lists
* elements of type-parameter-lists

All attributes are specified in *attribute sections*. A valid attribute section is made up of an opening and closing square bracket ([ and ]), inside of which is a list of attributes separated by commas (the list can contain one or more attributes).

For example:

[A ,B]

It is important to note that neither the order in which the attributes are specified nor the order in which the sections in a program entity are arranged has any significance whatsoever. This means that the fol- lowing are equivalent:

[A, B]

[B, A]

1. [B] [B][A]

The syntax of attribute specification is shown as follows:

global-attributes:

global-attribute-sections

global-attribute-sections: global-attribute-section

global-attribute-sections global-attribute-section

global-attribute-section:

[ global-attribute-target-specifier attribute-list ] [ global-attribute-target-specifier attribute-list , ]

global-attribute-target-specifier: global-attribute-target :

global-attribute-target: identifier

keyword

attributes:

attribute-sections

attribute-sections: attribute-section

attribute-sections attribute-section

attribute-section:

[ attribute-target-specifieropt attribute-list ] [ attribute-target-specifieropt attribute-list , ]

attribute-target-specifier: attribute-target :

attribute-target: identifier keyword

attribute-list: attribute

attribute-list , attribute

attribute:

attribute-name attribute-argumentsopt

attribute-name: type-name

attribute-arguments:

( positional-argument-listopt )

( positional-argument-list , named-argument-list )

( named-argument-list )

positional-argument-list: positional-argument

positional-argument-list , positional-argument

positional-argument:

attribute-argument-expression

named-argument-list: named-argument

named-argument-list , named-argument

named-argument:

identifier = attribute-argument-expression

attribute-argument-expression: expression

An attribute is made up of:

* An attribute-name
* An optional list of positional and named arguments

Any positional attributes must be listed before any named arguments. A positional attribute is made up of:

* An attribute-argument-expression, followed by
* A name, followed by
* An equal sign, followed by
* An attribute-argument-expression

The order of named arguments is not important and does not convey any significance.

When an attribute is placed at the global level, a global-attribute-target-specifier is manda- tory. The only standardized global-attribute-target name is assembly.

The only standardized attribute-target names are:

* event — An event
* field — A field
* method — A constructor, finalizer, method, operator, property get and set accessors, the event

add and remove accessors

* param — A property set of accessors, event add and remove accessors, and a parameter in a con- structor, method, or operator
* property — A property
* return — A delegate, method, property, or operator
* type — A class, delegate, enum, interface, or struct
* typevar — A type parameter

An expression E is only considered an attribute-argument-expression if all of the following statements are true:

* The type of E is an attribute parameter type.
* If, when a compile-time error occurs, the value of E can be resolved to:
  + A typeof-expression
  + A constant value
  + A one-dimensional array consisting of attribute-argument-expressions

##### Attribute Instances

An attribute instance is used to represent an attribute during runtime. An attribute is defined with:

* An attribute class
* Positional arguments
* Named arguments

An attribute instance is an instance of the attribute class that has been initialized with positional and named arguments.

Attribute Compilation

The compilation of an attribute with attribute class T, positional-argument-list P and named- argument-list N, is made up of the following steps:

* + Follow the compile-time processing steps for compiling an object-creation-expression of the form new T(P). These steps will either determine an instance constructor C on T that can be invoked at runtime or result in a compile-time error.
  + If C does not contain any public accessibility, this will result in a compile-time error.
  + For each named-argument Arg in N:
    - Name will be the identifier of the named-argument Arg.
    - Name must identify a nonstatic read-write public field or property on T. If no such field or property exists, this results in a compile-time error.

Keep the following information in mind for runtime instantiation of the attribute:

* + Attribute class T
  + Instance constructor C on T
  + The positional-argument-list P
  + The named-argument-list N

Runtime Retrieval of Attribute Instances

Here are the steps necessary to retrieve an attribute instance represented by T, C, P, and N and associated with E at runtime from an assembly A:

* + Follow the runtime processing steps for executing an object-creation-expression of the form new T(P), using the instance constructor C as determined at compile time. This will result in an instance O of T or in a compile-time error.
  + For each named-argument Arg in N, the following are carried out in order:
  1. Let Name be the identifier of the named-argument Arg. If Name does not identify a non- static public read-write field or property on O, this will result in an exception being thrown.
  2. Let Value be the result of evaluating the attribute-argument-expression of Arg.
  3. If Name identifies a field on O, this field should be set to Value.
  4. Else, Name identifies a property on O and this should be set to Value.
  5. The result is O, an instance of the attribute class T that has been initialized that has

positional-argument-list P and named-argument-list N.

##### Reserved Attributes

The following attributes have the stated effect on the code:

* System.AttributeUsageAttribute — Used to describe the ways an attribute class can be used

[AttributeUsageAttribute(AttributeTargets.Class, Inherited=true)]

* System.ObsoleteAttribute — Used to mark a member obsolete

[AttributeUsageAttribute(AttributeTargets.Class| AttributeTargets.Struct|

AttributeTargets.Enum| AttributeTargets.Constructor| AttributeTargets.Method| AttributeTargets.Property| AttributeTargets.Field| AttributeTargets.Event| AttributeTargets.Interface| AttributeTargets.Delegate, Inherited=false)]

* System.Diagnostics.ConditionalAttribute — A multiuse attribute class used to define conditional attribute classes and conditional methods

[AttributeUsageAttribute(AttributeTargets.Class| AttributeTargets.Method, AllowMultiple=true)]

The Conditional Attribute

The attribute Conditional enables the definition of:

* Conditional methods
* Conditional attribute classes

Conditional Methods

A method that has a Conditional attribute is known as a *conditional method*. Every conditional method is linked with the conditional compilation symbols declared by the Conditional attributes.

A conditional method has the following constraints:

* The conditional method has to be a method in a class-declaration or struct-declaration; otherwise, a compile-time error is generated.
* A conditional method cannot have an override modifier.
* A conditional method cannot be an implementation of an interface method.
* The conditional method has to have a return type of void.

*A compile-time error will be generated if any conditional methods are used in a* delegate-creation- expression*.*

Conditional Attribute Classes

An attribute class that has one or more Conditional attributes is known as a *conditional attribute class*. It therefore stands to reason that a conditional attribute class is associated with the conditional compilation symbols (looked at in Chapter 4) declared in its Conditional attributes.

[Conditional(.DEBUG.)]

public static void Help(String str)

{

Console.WriteLine(str);

}

The Obsolete Attribute

The Obsolete attribute is used to mark any types or members that shouldn’t be used. When a type or member is marked with the Obsolete attribute, using it generates a warning at compile time. The com- piler also generates a warning if:

* + No error parameters are supplied.
  + The error parameter provided has the value false. The compiler will generate an error if:
    - An error parameter is specified, and it has the value true. Here is an example of the Obsolete attribute in action:

Using System;

Namespace MyExample

{

Class MyAttribute

{

[Obsolete()]

public static void Test(string str)

{

Console.WriteLine(str);

}

public static void Test2(string str)

{

Console.WriteLine(str);

}

static void Main()

{

Test(“This is a test”); Console.ReadLine();

}

}

}

To get a custom message displayed, we would use the following code:

Using System;

Namespace MyExample

{

Class MyAttribute

{

[Obsolete(“The Test() method obsolete. Instead use Test2()”)]

public static void Test(string str)

{

Console.WriteLine(str);

}

public static void Test2(string str)

{

Console.WriteLine(str);

}

static void Main()

{

Test(“This is a test”); Console.ReadLine();

}

}

}

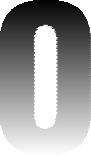
##### Summary

This chapter has looked at how to use a powerful feature of C# called attributes. Attributes are a way of defining declarative tags within the source code of a program.

The chapter began with a look at what attributes are and where they can be used, before looking at specifics of attributes, such as:

* Attribute classes
* Positional parameters
* Named parameters
* Attribute usage
* Specifying attributes
* Attribute instances
* Reserved attributes

In Chapter 20, you move on to look at generics.



### Generics

In this chapter we are going to look at generics and how they allow programmers to write clearer code that performs better. We’ll start by comparing generics in C# with templates in C++ before going on to look at the advantages of generics, followed by a detailed look at them.

Generics are, without a doubt, the most powerful feature introduced into C# 2.0, enabling the programmer to define type-safe data structures without having to define an actual data type. This has a number of advantages:

* + Greater performance
  + Code that is more readable

##### C# Generics vs. C++ Templates

A number of comparisons can be drawn between generics in C# and templates in C++. Here is a good way to think of the two:

* + Think of generics in C# as classes, except the former are a type parameter.
  + Think of C++ templates as macros, except the code for templates looks like the code for classes.

There are a couple of other important differences:

* + With C#, the instantiation of generics is done during runtime (when the program is being run) by the JIT compiler. The runtime is creating native code specifically for the type in question when it is needed. With templates in C++, all this is carried out at compile time or link time.
  + C# carries out strong type-checking when a generic is compiled, which guarantees that any operation carried out on a type parameter will work. With C++, there is none of this, which can lead to very generic error messages. In this way, C# generics can be thought of as strongly typed, whereas C++ templates are untyped or, at best, loosely typed.

Advantages of Generics

There are a number of advantages to using generics, some of which we’ve touched on already:

* Generics allow the specification of types at runtime.
* There is no need for boxing (the name given to converting a value to a reference type) or casting (explicitly converting between data types), which means greater performance.
* Fewer cryptic error messages and less debugging time
* Clearer, easier-to-understand code

Here is an example of generics in action. Here we have a generic call, Compare, that compares two items that have the same type and returns the largest or smallest, depending on which method is called in

the code:

public class Compare<ItemType, ItemType>

{

public ItemType Larger(ItemType info1, ItemType info2)

{

// Code goes here

}

public ItemType Smaller(ItemType info1, ItemType info2)

{

// Code goes here

}

}

##### Generic Class Declarations

A generic class declaration needs type arguments to be supplied so that runtime types can be formed when the MSIL code is processed by the JIT compiler.

The syntax for generic class declarations is shown as follows:

class-declaration:

attributesopt class-modifiersopt partialopt class identifier type-parameter-listopt class-baseopt type-parameter-constraints-clausesopt class-body ;opt

*A class declaration cannot provide* type-parameter-constraints-clauses *unless the declara- tion also supplies* type-parameter-list*.*

The rules governing generic classes are similar to those that govern nongeneric classes, and it is possible for generic class declarations to be nested within nongeneric class declarations.

Generic classes are referenced using a constructed type. For example, take the following generic class:

class List<T> {}

This could be accessed by a number of constructed types:

* + List<int>
  + List<T>
  + List<List<string>>

There are two types of constructed types:

* + **Open constructed types.** These use one or more type parameters. For example:

List<T>

* + **Closed constructed type.** These use no type parameters:

List<int>

Type Parameters

Type parameters are supplied in a class declaration, and each type parameter is a simple identifier that acts as a placeholder for a type argument supplied to create a constructed type. The actual type for the type parameter is supplied later in the code.

Compare this to a type argument that is a runtime type later substituted for the type parameter when a constructed type is created.

The syntax of type parameter lists is shown as follows:

type-parameter-list:

< type-parameters >

type-parameters:

attributesopt type-parameter

type-parameters , attributesopt type-parameter

type-parameter: identifier

Each type parameter found in a class declaration defines a specific name in the declaration space of that class. This means that a type parameter can’t have the same name as another type parameter or a mem- ber declared in the class.

*In addition, a type parameter cannot have the same name as the type itself.*

The scope of a type parameter on a class covers:

* + class-base
  + type-parameter-constraints-clauses
  + class-body

Be aware, though, that this scope does not extend to derived classes (which differs from the behavior of class members).

Type Parameter Differences

Because of the way type parameters work (in that they can be instantiated with different runtime argu- ments), they differ from other types in a few key ways. Here are a few examples of these differences:

* A type parameter cannot be used to declare a base class or interface.
* The rules for member lookup on type parameters depend on any constraints that might have been applied to the type parameter:
  + During member lookup, members declared in a class other than object hide members declared in interfaces.
  + During overload resolution of methods and indexers, if any applicable member was declared in a class other than object, all members declared in an interface are removed from the set of considered members.
* The literal null cannot be converted to a type given by a type parameter, unless the type param- eter is known to be a reference type (note that it is possible to use a default value expression).
* A new expression can only be used with a type parameter when the type parameter is con- strained by a constructor-constraint or the value type constraint.
* The available conversions for a type parameter depend on any constraints that might have been applied to the type parameter.
* A type parameter cannot be used anywhere within an attribute.
* A type parameter cannot be used in a member access or type name to identify either a static member or a nested type.
* In unsafe code, a type parameter cannot be used as an unmanaged type.

Instance Type

Every class declaration has an associated constructed type called an *instance type*. For generic class decla- rations, the instance type is created by forming a constructed type from the type declaration, with each type argument being a corresponding type parameter.

Because instance types use type parameters, they can only be used in code locations where the parame- ters are in scope (inside the class declaration itself).

The following code shows several class declarations:

class A<T>

{

class B {}

class C<X> {}

}

class D {}

The table that follows shows classes with their associated instance types:

|  |  |  |  |
| --- | --- | --- | --- |
| **Class** |  |  | **Instance Type** |
| class | A<T> |  | A<T> |
| class | B {} |  | A<T>.B |
| class | C<X> | {} | A<T>.C<X> |
| class | D {} |  | D |

Generic Class Members

All members of a generic class can make use of type parameters from any enclosing class. This can be done either directly or as part of a constructed type.

When a closed constructed type is used, each use of a type parameter is replaced with the runtime type argument supplied to the constructed type at runtime.

Here is an example of generic class members in action:

using System; class C<T>

{

private static int m\_NCount = 0; public C() { m\_NCount++; }

public int NCount { get { return m\_NCount; } }

}

class Program

{

static void Main()

{

C<int> c1 = new C<int>(); C<int> c2 = new C<int>(); C<string> c3 = new C<string>(); C<string> c4 = new C<string>(); C<string> c5 = new C<string>(); C<object> c6 = new C<object>();

Console.WriteLine( “C<int> : “ + c1.NCount.ToString() ); Console.WriteLine( “C<string>: “ + c3.NCount.ToString() ); Console.WriteLine( “C<object>: “ + c6.NCount.ToString() );

}

}

The output from this program is as follows:

C<int> : 2

C<string>: 3

C<object>: 1

Static Fields in Generic Classes

A static variable in a generic class declaration is shared with all instances of the same closed constructed type. It is not shared among instances of different closed constructed types.

These two rules apply whether the type of the static variable involves any type parameters or not.

Static Constructors in Generic Classes

A static constructor in a generic class is used to initialize static fields. It is also used to carry out initial- ization for each different closed constructed type resulting from that generic class declaration.

The type parameters of the generic type declaration are in scope within the body of the static constructor. A new closed constructed class type is initialized the first time that one of the following conditions exists:

* An instance of the closed constructed type is created.
* Any of the static members of the closed constructed type are referenced. New closed constructed type fields are created as follows:
* A new set of static fields for the closed constructed type is created.
* The static fields are initialized with default values.
* The static field initializers are executed.
* Finally, the static constructor is executed.

Access to Protected Members

Inside a generic class declaration, access to inherited, protected instance members is allowed through an instance of any class type constructed from the generic class.

Overloading in Generic Classes

The following in generic classes can be overloaded:

* methods
* constructors
* indexers
* operators

Here is an example of an overloaded method:

Public void functionName(int x, params int[] varParam); Public void functionName(int x);

When overloading, declared signatures have to be unique. However, even when signatures are unique, this doesn’t mean that substitution of type arguments can’t result in identical signatures.

Operators in Generic Classes

Generic class declarations can also define operators. The rules are the same as for nongeneric class decla- rations. The instance type of the class declaration is used to declare operators. The rules for this are as follows:

* + A unary operator will take a single parameter of the instance type.
  + Both unary ++ and -- operators will return the instance type or a type derived from the instance type.
  + A minimum of one of the parameters of a binary operator has to be of the instance type.
  + Either the parameter type or the return type of a conversion operator has to be an instance type.

No rule prevents you from declaring operators that will specify conversions that already exist as prede- fined conversions for some argument types. However, if conversions are specified where there are predefined conversions between two types, conversions specified by the code will be ignored, and pre- defined conversions will be used.

##### Generic Struct Declarations

A struct declaration can be used to define type parameters and their associated constraints:

struct-declaration: attributesopt struct-modifiersopt partialopt

struct identifier

type-parameter-listopt struct-interfacesopt

type-parameter-constraints-clausesopt struct-body ;opt

##### Generic Interface Declarations

An interface declaration can be used to define type parameters and their associated constraints:

interface-declaration: attributesopt interface-modifiersopt partialopt

interface identifier

type-parameter-listopt interface-baseopt

type-parameter-constraints-clausesopt

interface-body ;opt

Each type parameter in an interface declaration defines a name in the declaration space of the interface in question.

The scope of a type parameter on an interface includes:

* interface-base
* type-parameterconstraints-clauses
* interface-body

A type parameter can be used as a type wherever it has scope.

##### Explicit Interface Member Implementations

Explicit interface member implementations work with constructed interface types in much the same way as they do with simple interface types. An explicit interface member implementation will be qualified by an interface type used to indicate which interface is being implemented.

This type can be either a simple interface or a constructed interface.

##### Generic Delegate Declarations

A delegate declaration can be used to define type parameters and their associated constraints:

delegate-declaration: attributesopt delegate-modifiersopt delegate

return-type identifier

type-parameter-listopt

( formal-parameter-listopt )

type-parameter-constraints-clausesopt ;

Each type parameter in a generic delegate declaration is used to define a name in a declaration space that will be associated with that delegate declaration.

The scope of a type parameter in a delegate declaration includes the following:

* return-type
* formal-parameter-list
* type-parameter-constraints clauses

Like all other generic type declarations, type arguments are used to create a constructed delegate type.

##### Constructed Types

A generic type declaration on its own is an unbound generic type. This is used as a template from which many different types can be created by applying type arguments.

The type arguments, described in the following section, are written inside angle brackets (< and >), which immediately follow the name of the generic type declaration.

Any type named by one or more type argument is called a constructed type.

Type Arguments

Every argument that appears in a type argument list is merely a type:

type-argument-list:

< type-arguments >

type-arguments: type-argument

type-arguments , type-argument

type-argument: type

Type arguments can be either constructed types or type parameters.

Open and Closed Types

Every type can be classified as either an *open type* or a *closed type*. An open type uses type parameters:

* + A type parameter defines an open type.
  + An array type is an open type only if the element types are an open type.
  + A constructed type is an open type only if one or more of the type arguments are an open type. An example of open types is:

Stack<T>

A closed type is not an open type. Examples of closed types are:

Stack<int> Stack<Stack<int>>

Members of Constructed Types

The noninherited members of any constructed type can be derived by substituting each type parameter in the member declaration for the corresponding type argument of the constructed type.

The inherited members of any constructed type can be obtained in a similar way.

To do this, all the members of the immediate base class have to be determined. If the base class is itself a constructed type, this might mean that the current rule has to be applied recursively. After this, each of the inherited members is transformed by substituting, for each type parameter in the member declara- tion, the corresponding type argument of the constructed type.

Using Alias Directives

Aliases can name a closed constructed type, but they cannot name a generic type declaration without supplying type arguments.

Generic Methods

A generic method is a method where the declaration includes a type-parameter-list. Generic meth- ods can be declared inside the following declarations:

* class
* struct
* interface

These declarations can be either generic or nongeneric. Here is a code example of a generic method:

public T Test<T>(T val1, T val2) where T : IComparable { T retVal = val2;

if (val2.CompareTo(val1) < 0) retVal = val1;

return retVal;

}

When a generic method is declared inside a generic type declaration, the body of the method can refer both to the type parameters of the method and the type parameters of the containing declaration.

The type-parameter-list and type-parameter-constraints-clauses of a generic method decla- ration have the same syntax and purpose as in a generic type declaration.

The method’s type parameters are in scope throughout the method declaration and can be used to form types throughout that scope in the following:

* + return-type
  + method-body
  + type-parameter-constraints-clauses

The name of a method type parameter cannot be the same as the name of an ordinary parameter in the same method.

Generic Method Signatures

For the purposes of signature comparisons, any type-parameter-constraints-clauses present are ignored. Similarly, the names of the method’s type parameters are also ignored. Not ignored are the number of generic type parameters and the ordinal positions of type parameters in left-to-right ordering.

Virtual Generic Methods

Generic methods can be declared using the following modifiers:

* + abstract
  + virtual
  + override

Signature matching rules are used when matching methods to a particular override and interface imple- mentation. Whenever a generic method is used to override another declared in a base class, that method cannot specify any type-parameter-constraints-clauses, because constraints are inherited from the method being overridden. The same is true of interface implementation.

Calling Generic Methods

A generic method can either specify an argument type list itself or can instead fall back on type infer- ence, described in the following section, and allow that to determine type arguments.

Be mindful that allowing generic methods to use type inference can sometimes lead to the code being hard to follow and understand.

***Inference of Type Arguments***

When a generic method is called without type arguments, the process called type inference is used to infer the type arguments for the particular calling of the method. This is carried out at compile time. The advantage of using type inference is that it enables code to be written that is more concise.

It is important to note that if type inference fails, a compiler error will not occur during compilation, but the method will not take part in any overload resolution — and this can cause a compiler error later when no methods are found.

Let’s assume this argument has type T and the corresponding parameter has type P. Type inferences are worked out as follows:

* No inference is made if any of the following are true:
  + P does not involve any method type parameters.
  + The argument is an anonymous method.
  + The argument is a method group.
  + The argument has the null type.
* If P and A are array types of the same rank, replace A and P with the element types of A and P, and repeat this step.
* If P is a method type parameter, type inference succeeds for this argument, and A is the type inferred for that type parameter.
* If P is an array type and A is not an array type of the same rank or an instantiation of IList<>, ICollection<>, or IEnumerable<>, type inference fails for the generic method.
* If P is an array type and A is an instantiation of IList<>, ICollection<>, or IEnumerable<>, replace A and P with the element types of A and P, and repeat this step.
* Otherwise, P will be a constructed type:
  + If, for each method type parameter MX found in P, one type (and only one type) TX can be determined so that replacing each MX with each TX produces a type to which A can be changed by a standard implicit conversion, inferencing succeeds for this argument, and each TX is the type inferred for each MX.
  + Method type parameter constraints, if any, are ignored for the purpose of type inference.
  + If, for a particular MX, no TX exists, or there is more than one TX, type inference will fail for the generic method.

Type inference is said to have been successful if both of the following are true:

* Each type parameter of the method had a type argument inferred for it.
* For every type parameter, all of the inferences for that type parameter infer the same type argument.

##### Where Generics Aren’t Used

The following don’t have type parameters:

252

* Constructors
* Events
* Finalizers
* Indexers
* Operators
* Properties

While the preceding items don’t have type parameters, this doesn’t stop them from appearing as generic types, and they can use any type parameters from the enclosing type.

##### Constraints

Generic type and method declarations can also optionally specify one or more type parameter con- straints by including a type-parameter-constraints-clauses in the declaration:

type-parameter-constraints-clauses: type-parameter-constraints-clause type-parameter-constraints-clauses type-parameter-constraints-clause

type-parameter-constraints-clause:

where type-parameter : type-parameter-constraints

type-parameter-constraints: primary-constraint secondary-constraints constructor-constraint

primary-constraint , secondary-constraints primary-constraint , constructor-constraint secondary-constraints , constructor-constraint

primary-constraint , secondary-constraints , constructor-constraint

primary-constraint: class-type class

struct

secondary-constraints: interface-type type-parameter

secondary-constraints , interface-type secondary-constraints , type-parameter

constructor-constraint: new ( )

Each type-parameter-constraints-clauses consists of:

* + The token where, followed by
  + The name of a type parameter, followed by
  + A colon and the list of constraints for that type parameter

There can be no more than one where clause for each type parameter, and where clauses can be listed in any order.

*Note that the* where *token is not a keyword.*

The list of constraints given in a where clause can include any of the following components, in this order:

* A single primary constraint
* One or more secondary constraints
* Finally, the constructor constraint, new()

A primary constraint can be any of the following:

* A class type
* The reference type constraint class
* The value type constraint struct A secondary constraint can be either:
* A type parameter
* An interface type

The reference type constraint specifies that a type argument used for the type parameter has to be a reference type. The following all satisfy this constraint:

* Array type
* Class type
* Delegate type
* Interface type
* Type parameters that are reference types

The value type constraint specifies that a type argument used for the type parameter has to be a value type. The following all satisfy this constraint:

* Enum type
* Any non-nullable struct types
* A type parameter having the value type

If a constraint is a class type, a type parameter, or an interface type, it is a type that specifies a minimal “base type” that every type argument used for that type parameter will be able to support.

A class-type constraint has to satisfy the following rules:

* The type has to be a class type.
* The type cannot be sealed.
* The type cannot be one of the following:
  + System.Array
  + System.Delegate
    - System.Enum
    - System.ValueType
* The type cannot be object.
* Only one constraint for any specified type parameter can be a class type. A type specified as an interface-type constraint has to satisfy the following rules:
* The type has to be an interface type.
* A type cannot be specified more than once in a given where clause.

The constraint can use any of the type parameters of the associated type or method declarations as part of the constructed type. It can also use the type being declared.

A type specified as a type-parameter constraint has to fulfill the following rules:

* A type cannot be specified more than once in any given where clause.
* The type has to be a type parameter.

The effective base class of a type parameter T is defined as follows:

* If T doesn’t have any primary constraints or type parameter constraints, its effective base class is

object.

* If T has the value type constraint, its effective base class is System.ValueType.
* If T has a class-type constraint C but doesn’t have any type-parameter constraints, its effec- tive base class is C.
* If T doesn’t have a class-type constraint but has one or more type-parameter constraints, its effective base class is the most encompassed type in the set of effective base classes of its type-parameter constraints.
* If T has both a class-type constraint and one or more type-parameter constraints, its effec- tive base class is the most encompassed type in the set that consists of the class-type con- straint of T and the effective base classes of the type-parameter constraints.

The effective interface set of a type parameter T is defined as follows:

* If T doesn’t have any secondary constraints, its effective interface set is empty.
* If T has interface-type constraints but doesn’t have type-parameter constraints, its effective interface set is its set of interface-type constraints.
* If T doesn’t have any interface-type constraints but does have type-parameter constraints, its effective interface set is the union of the effective interface sets of its type-parameter con- straints.
* If T has both interface-type constraints and type-parameter constraints, its effective inter- face set is the union of its set of interface-type constraints and the effective interface sets of its type-parameter constraints.

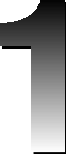
##### Summary

In this chapter you looked at one of the most powerful features of C# — generics. This new feature is similar to templates in C++, but there are some key differences:

* Instantiation of generics is performed during runtime.
* C# carries out strong type-checking when a generic is compiled.

This allows for a number of advantages, including specification of types at runtime and the reduced need for boxing and casting operations that can be system intensive.

In Chapter 21, you look at iterators and how they are used in C#.

### Iterators

In this chapter we are going to take a look at how iterators are used in C#.

An iterator provides C# with a way of implementing a function whose return type is either:

* + An enumerator interface
  + An enumerable interface

*The difference between these is described later in this chapter.*

The function member then returns an ordered sequence of values yielded by the operator. Take a look at the following code:

using System;

using System.Collections;

public class Months : IEnumerable{ string[] m\_Names;

public Months(params string[] Names){ m\_Names = new string[Names.Length]; Names.CopyTo(m\_Names, 0);

}

public IEnumerator GetEnumerator(){ foreach (string s in m\_Names)

yield return s;

}

}

class Program{

static void Main(string[] args){

Months arrMonths = new Months(“Jan”,

“Feb”,

“Mar”,

“Apr”,

“May”,

“Jun”,

“Jul”,

“Aug”,

“Sep”,

“Oct”,

“Nov”,

“Dec”);

foreach (string s in arrMonths) Console.WriteLine(s);

Console.ReadLine();

}

}

The output of the preceding code is as follows:

Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec

Iterators are implemented using yield statements. These yield statements can only be used with meth- ods where the return type is an enumerator interface.

In the preceding example, the GetEnumerator makes the m\_Names that you see in the foreach loop an enumerable type.

##### Iterator Block

An iterator block is a block of code that will, when processed, yield a sequence of values ordered in a particular fashion. You can spot an iterator block in code and tell it apart from ordinary statements by looking for the yield statement that will appear one or more times in the block.

Following is an example of an iterator block:

public class Months : IEnumerable{ string[] m\_Names;

public Months(params string[] Names){ m\_Names = new string[Names.Length]; Names.CopyTo(m\_Names, 0);

}

public IEnumerator GetEnumerator(){ foreach (string s in m\_Names)

yield return s;

}

}

There are two types of yield statements:

* + **The** yield return **statement.** This statement produces the next value of the iteration:

public IEnumerable GetEnumerator()

{

for (int x=0; x<itemArray.Length; x++)

yield return itemArray[x];

}

* + **The** yield break **statement.** This statement indicates that the iteration is complete:

public IEnumerable GetShortEnumerator(int l)

{

for (int x=0; x<itemArray.Length; x++)

{

yield return itemArray[x]; if (x==l)

yield break;

}

}

Iterator blocks are, grammatically speaking, just normal blocks of code. While they have an effect on code semantics, iterator blocks should not be considered different from other blocks of code.

Iterator Blocks and Compile-time Errors

Doing the following will result in a compile-time error:

* + Having a function member implemented where the parameter list specifies any ref or out

parameters

* + Having a return statement appear in the iterator block (yield return is allowed, though)
  + Having any iterator block that contains an unsafe context (see Chapter 22 for more details)

Enumerator Interfaces

Enumerator interfaces are the nongeneric interface System.Collections.IEnumerator. The System.Collections.IEnumerator interface also includes all instances of the generic interface System.Collections.Generic.IEnumerator<T>.

Enumerable Interfaces

Enumerable interfaces are the nongeneric interface System.Collections.IEnumerable. The System.Collections.IEnumerable interface also includes all instances of the generic interface System.Collections.Generic.IEnumerable<T>.

259

Yield Type

An iterator block will output a sequence of values. These values will all have the same type. The type is called the *yield type* of the iterator block.

* Function members that return IEnumerator or IEnumerable will have the yield type of object.
* Function members that return IEnumerator<T> or IEnumerable<T> will have the yield type of T.

This

When placed inside an instance member of a class, a this expression is classed as a value. The type of this value will be the class within which it is found. The value is a reference to the object for the member that was invoked.

When this is found within an iterator block of an instance member of a struct, it is classed as a variable. The type of the variable is the struct where it occurs.

##### Enumerator Objects

Function members in iterator blocks that return enumerator interface types behave differently from stan- dard function members. When the function member is invoked, the code inside the iterator block is not executed straight away. Instead, an enumerator object that encapsulates the code contained in the itera- tor block is created and returned. Execution of the code occurs when the MoveNext method of the object is invoked.

The following are characteristics of the enumerator object:

* The enumerator object implements IEnumerator and IEnumerator<T> (where T is the yield type).
* Enumerator objects implement System.IDisposable.
* Enumerator objects are initialized with a copy of any argument values and instance values passed to the function members.
* There are four states for enumerator objects:
  + Before (the initial state)
  + Running
  + Suspended
  + After

The MoveNext Method

The MoveNext method is used to encapsulate the code of the iterator block. By invoking the MoveNext method, you are executing the code that is inside the iterator block. Invoking the code with MoveNext also sets the Current property of the enumerator object as appropriate.

What happens when MoveNext is invoked depends on the state of the enumerator before it is invoked.

* + - State = before:
      * The state is changed to running.
      * The parameters of the iterator block are initialized to the argument values and instance value saved when the enumerator object was initialized.
      * The iterator code block is executed, and this continues until interrupted.
    - State = running:
      * Action is unspecified.
    - State = suspended:
      * The state is changed to running.
      * All local variables and parameters (including this) are reset to the values saved when the execution was suspended.
      * Execution of the code that immediately follows the yield return that caused the suspen- sion in the iteration block is resumed, and the code execution continues until interrupted.
    - State = after:
      * MoveNext returns false.

Here’s an example of the MoveNext method in action:

public bool MoveNext()

{

index++;

if (index >= x.strings.Length)

{

return false;

}

else

{

return true;

}

}

Execution Interruptions

There are four ways that execution of the iteration block with MoveNext can be interrupted:

* + - When the yield return is encountered:
      * The expression in the statement is evaluated and implicitly converted to the yield type. It is then assigned to the current property of the enumerator object.
      * The execution of the code in the iterator is then suspended. All local variables and parame- ters are saved (including this). The location of the yield return statement is also saved.
      * The state of the enumerator object is changed to suspended.
      * The MoveNext method returns a true to the caller, which signals that the iteration has advanced to the next value. 261
* When a yield break statement is encountered:
  + If the yield break statement appears inside a try block, any associated finally blocks are executed.
  + The state of the enumerator object is changed to after.
  + The MoveNext method returns a false to the caller to indicate that iteration has completed.
* When the iteration body ends:
  + The state of the enumerator object is changed to after.
  + The MoveNext method returns a false to the caller to indicate that iteration has completed.
* An exception is thrown that propagates out of the iteration code block:
  + Any finally blocks are executed as the exception propagates.
  + The state of the enumerator object is changed to after.
  + Exception propagation continues to the caller of the MoveNext method.

The Current Property

The Current property of an enumerator object is affected by yield return statements in the iterator block.

The value of Current is dependant on the state of the object:

* When an enumerator object is in the suspended state, the value of Current is the value set by the previous call to MoveNext.
* When an enumerator object is in the before, running, or after states, the result of accessing

Current will be unspecified.

Here is an example of the Current property in action:

public class MyEnumerator<T> : IEnumerator<T>

{

public T Current

{

get

{

// code goes here

}

}

}

The Dispose Method

Dispose is used as a clean-up method and takes the enumerator object to the after state.

The state of the Dispose method depends on the enumeration object as detailed below:

* + State of enumeration object = before:
    - Invoking Dispose changes state to after.
  + State of enumeration object = running:
    - Invoking Dispose is unspecified.
  + State of enumeration object = suspended:
    - State is changed to running.
    - Any finally code blocks are executed (if the last yield return statement was a yield break statement). Any exceptions thrown will propagate out to the caller of the Dispose method and the state is changed to after.
    - State is changed to after.
  + State of enumeration object = after:
    - Invoking Dispose will have no effect. Here is an example of the Dispose method in action:

{

if(x != null) ((IDisposable)x).Dispose();

}

##### Enumerable Objects

When a function member that returns an enumerable interface type is implemented using an iterator block, invoking the function member does not execute the code in the iterator code block. Instead, an enumerable object is created, and this is returned.

The iterator code block is encapsulated by the enumerable object’s GetEnumerator method. The execu- tion of the code inside the iterator block happens when the MoveNext method of the enumerator object is invoked.

The following are characteristics of the enumerable object:

* + The enumerator object implements IEnumerable and IEnumerable<T> (where T is the yield type).
  + Enumerator objects are initialized with a copy of any argument values and instance value passed to the function members.

GetEnumerator Method

An enumerable object provides an implementation of the GetEnumerator methods of both the

IEnumerable and IEnumerable<T> interfaces.

263

The two GetEnumerator methods both acquire and return an available enumerator object. The enumer- ator object is initialized with the argument values and instance value saved when the enumerable object was initialized.

The following example shows the GetEnumerator method in action. Here the method will return either an enumerator or an enumerable class for an ordered list of items. The order is preserved using a yield statement:

public IEnumerable GetEnumerator()

{

for (int x=0; x<itemArray.Length; x++) yield return itemArray[x];

}

This next example uses a yield break to indicate that the last item has been yielded:

public IEnumerable GetShortEnumerator(int l)

{

for (int x=0; x<itemArray.Length; x++)

{

yield return itemArray[x]; if (x==l)

yield break;

}

}

##### Summary

In this chapter you examined iterators in C# and how they can be used to return an ordered sequence of values. You looked in some detail at how the yield return and yield break statements offer flexibil- ity in coding and how the four states of the enumerator objects provide great flexibility when coding.

In Chapter 22, you examine safe and unsafe code.



### Unsafe Code

If you’ve come from a C++ background, you might have noticed one feature of C++ that has so far been absent in C# — pointers.

In C#, the majority of memory management tasks that a C++ programmer would need to worry about are taken care of automatically. The thorough garbage collection in C# (and the .NET Framework), along with the extensive use of references, means that the C# programmer can write powerful code yet remain totally oblivious to memory management.

However, there are times when it would be useful to have direct access to the memory in order to be able to write code that is more powerful and versatile than regular code. This kind of code is known as *unsafe code*.

##### What is Unsafe Code?

While it is true that every pointer type construct found in C++ (or C, for that matter) has a compa- rable reference type in C#, there are times when direct access to a pointer type is either useful or needed — for example, when you want to write code that interacts with the operating system or want to access a device that has been mapped in memory.

C# was designed to be a safe and easy-to-use language. One of the goals of C# was to eliminate the need to write code that might cause problems (for example, in C++ it was possible to have variables that weren’t initialized or to write code that indexed arrays out of bounds). C# was designed with safe code in mind. However, its designers also recognized that there are times when a programmer might want to write unsafe code. In unsafe code, it is possible for the programmer to both declare and operate on pointers. Also possible is conversion between pointers and integral types, to retrieve the address of a variable and much more.

*Unsafe code is not executed under the full management of the Common Language Runtime.*

It’s important to note that unsafe code is still safe. But to be safe, all unsafe code has to be clearly marked with the unsafe modifier. This way, unsafe features of C# cannot be used accidentally. This safety feature is used by C# to control how it is used.

##### Advantages and Disadvantages of Unsafe Code

There are a number of advantages and disadvantages to using unsafe code in C#. The following sections provide a brief rundown of the pros and cons.

Advantages of Unsafe Code

There are a number of advantages to using unsafe code.

* First and foremost, the main advantages are performance and flexibility. Using pointers is the fastest and more efficient way to access and manipulate data.
* Using unsafe code allows you to retrieve the memory addresses of data by using pointers. There is no way to do this using safe code.
* Unsafe code also allows C# programs to maintain compatibility with old Windows APIs or third-party DLL files that make use of pointers. There are ways of accomplishing this without pointers (such as using DLLImport declarations), but it’s usually simpler to use pointers.

Disadvantages of Unsafe Code

Using unsafe code isn’t all advantages. There are some big disadvantages too.

* Using unsafe code leads to code that is more complex than regular C# code.
* Unsafe code is harder to use than safe code. Using unsafe code makes it easy to do things that aren’t good, like overwrite variables, access memory areas that don’t contain data, or cause a stack overflow.
* Unsafe code is a lot harder to debug.
* Pointers are not forgiving. Code that makes use of pointers is much more likely to hang or crash than safe code.

##### Unsafe Code Contexts

Unsafe code is present in C# in certain contexts only, called unsafe contexts. The unsafe modifier has to be used in the declaration of the type or member.

Following are the rules governing unsafe contexts:

* Declarations of a delegate, class, interface, or struct can include an unsafe modifier.
* Declarations of an event, field, finalizer, indexer, instance constructor, method, operator, prop- erty, or static constructor can include an unsafe modifier.
  + An unsafe-statement is a way that the programmer can specify an unsafe context within a block:

class-modifier:

...

unsafe

struct-modifier:

...

unsafe

interface-modifier:

...

unsafe

delegate-modifier:

...

unsafe

field-modifier:

...

unsafe

method-modifier:

...

unsafe

property-modifier:

...

unsafe

event-modifier:

...

unsafe

indexer-modifier:

...

unsafe

operator-modifier:

...

unsafe

constructor-modifier:

...

unsafe

finalizer-declaration:

attributesopt externopt unsafeopt ~ identifier ( ) finalizer-body attributesopt unsafeopt externopt ~ identifier ( ) finalizer-body

static-constructor-modifiers: externopt unsafeopt static unsafeopt externopt static

externopt static unsafeopt unsafeopt static externopt static externopt unsafeopt static unsafeopt externopt

embedded-statement:

...

unsafe-statement

unsafe-statement: unsafe block

##### Pointer Basics

Let’s cover some pointer basics. If you’re already familiar with C++, you can skip this introduction.

Pointers are variables that hold the addresses of other variables. A simple example is if variable x

contains the address of y, then x is said to point to y.

Once a pointer points to a variable, the value of the variable can be changed or retrieved through the pointer. Operations carried out through pointers are sometimes referred to as indirection.

The general form that a pointer variable declaration takes is:

type\* varname

Here type is the pointer’s base type, which must be a nonreference type, which means that you can’t declare a pointer to a class object. Note that \* must follow the type name. Also, varname is the name of the pointer variable.

To declare a variable var1 to be a pointer to an int, the following declaration is used:

int\* var1;

A declaration statement, following a type name with a \*, creates a pointer type. In C#, the \* is distribu- tive and is the declaration. The following declaration declares two variables:

int\* var1, var2;

Void Pointers

If you want to declare a pointer but not specify a type for it, it needs to be declared as a void pointer.

void \*var1;

Pointer Operators

Let’s take a look at two operators used with pointers — the & and \* operators.

& is a unary operator used to return the memory address for the operand:

int\* var1; int num = 7;

var1 = &num;

In this example, var1 contains the memory address for the variable num. This address will be the loca- tion of the variable in the computer’s memory. It’s important to note that this variable has nothing at all to do with the value of the variable num.

The operations carried out by & can be thought of as returning the memory address of the operand.

The \* operator is the compliment of the & operator. It is a unary operator that refers to the value of the variable located at the address specified by the operand.

int var2 = \*num;

Here the value of num is placed into the variable var2.

\* can also be used on the left side of the assignment:

\*num = 7:

Here the value 7 is put into the address for num.

##### Unsafe in Action

Any code that makes use of pointers has to be marked unsafe. This is done using the unsafe keyword. Individual statements can be marked unsafe, or entire methods can be marked unsafe, depending on how much unsafe code is used.

Take a look at the following example:

using System; class UnsafeClass

{

unsafe public static void Main()

{

int var1 = 7; int\* var2; var2 = &var1;

Console.WriteLine( “Initial value is “ + \*var2 );

\*var2 = 10;

Console.WriteLine( “New value is “ + \*var2); Console.ReadLine();

}

}

This code contains some interesting points worth highlighting. Here Main() is marked as unsafe:

unsafe public static void Main()

Here a pointer is created:

int\* var2;

The address of var1 is placed in the pointer var2: var2 = &var1;

The initial value output is assigned:

Console.WriteLine( “Initial value is “ + \*var2 );

Now the value 10 is assigned to the variable via the pointer created:

\*var2 = 10;

A new value is output:

Console.WriteLine( “New value is “ + \*var2);

If this code were compiled and run, the output would be as follows:

Initial value is 7 New value is 10

Using the fixed Modifier

The fixed modifier is often used when working with pointers. It is used to prevent a managed variable from being moved by the garbage collector. This is needed, for example, when a pointer refers to a field in a class object.

Since 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. The fixed modifier is a way to prevent this from happening.

Here is a general form of fixed:

fixed ( type\* p = &var1 )

{

// use fixed object

}

Here, p is a pointer being assigned the address of a variable. The object will remain at the current mem- ory location until the block of code has executed.

Note that 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 in action:

using System; class Test

{

public int number; public Test(int x)

{

number = x;

}

}

class FixedExample

{

unsafe public static void Main()

{

Test test=new Test(21);

fixed ( int\* pointer1 = &test.number)

{

Console.WriteLine( “Initial value is “ + \*pointer1);

\*pointer1 = 7;

Console.WriteLine( “New value is “ + \*pointer1); Console.Read();

}

}

}

In this example, fixed prevents test from being moved. Because the pointer points to test.number, if

test were moved, the pointer would point to an invalid location.

Let’s take a look at the highlights of this code. Here we are declaring a class called Test for use.

class Test

{

public int number; public Test(int x)

{

number = x;

}

}

unsafe public static void Main()

Here fixed is used to put the address of test.number into the pointer:

fixed ( int\* pointer1 = &test.number)

We now output the initial value to the screen:

Console.WriteLine( “Initial value is “ + \*pointer1);

A new number is now assigned via the pointer that was created:

\*pointer1 = 7;

An altered value is now displayed:

Console.WriteLine( “New value is “ + \*pointer1);

The output from this program will be as follows:

Initial value is 21 New value is 7

##### sizeof Operator

The sizeof operator is interesting to use. It can be used to return the number of bytes occupied by a data type.

The following is an example of the sizeof operator in action:

unsafe

{

Console.WriteLine(“bool: {0}”, sizeof(bool));

Console.WriteLine(“byte: {0}”, sizeof(byte));

Console.WriteLine(“sbyte: {0}”, sizeof(sbyte));

Console.WriteLine(“short: {0}”, sizeof(short));

Console.WriteLine(“ushort: {0}”, sizeof(ushort));

Console.WriteLine(“int: {0}”, sizeof(int));

Console.WriteLine(“uint: {0}”, sizeof(uint));

Console.WriteLine(“long: {0}”, sizeof(long));

Console.WriteLine(“ulong: {0}”, sizeof(ulong));

Console.WriteLine(“char: {0}”, sizeof(char));

Console.WriteLine(“float: {0}”, sizeof(float));

Console.WriteLine(“double: {0}”, sizeof(double));

Console.WriteLine(“decimal: {0}”, sizeof(decimal));

}

The output from this code is as follows:

bool: 1

byte: 1

sbyte: 1

short: 2

ushort: 2

int: 4

uint: 4

long: 8

ulong: 8

char: 2

float: 4

double: 8

decimal: 16

##### Using stackalloc

The keyword stackalloc instructs the runtime to allocate a portion of memory on the stack. It requires two things:

* The type
* The number of variables you’re allocating to the stack

For example, if you want to allocate enough memory to store five floats, you can write the following:

float \*pointerfloat = stackalloc float [5];

To allocate enough memory to store 21 shorts:

short \*pointershort = stackalloc short [21];

It is important to remember that stackalloc simply allocates memory. It doesn’t initialize it to any value. The advantage of stackalloc is the ultrahigh performance it offers, and it is left up to you to ini- tialize the memory locations that were allocated. One useful application of stackalloc is in creating arrays directly in the stack, which is far more efficient than arrays that are objects instantiated from System.Array, which are stored in the heap.

##### Compiling Unsafe Code

If you’ve tried to compile any of the preceding unsafe code, you will have received an error like this:

error CS0227: Unsafe code may only appear if compiling with /unsafe

To compile unsafe code using the command-line compiler, you will need to add the /unsafe argument:

csc test.cs /unsafe

This will allow the code to be compiled. To compile the code under Visual Studio .NET, you will need to go to the project property page and set Allow Unsafe Code Blocks to True in Configuration properties > .

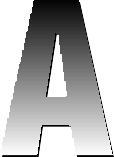
##### Summary

In this chapter you looked at unsafe code in C# and how it allows you to use pointers in C# in a way that C++ programmers will be comfortable and familiar with.

You also looked at what unsafe code is and the advantages and disadvantages of using unsafe code in programs.

You then moved on to look at the contexts where unsafe code can be used, before looking at the basics of using pointers in code.

Finally, you looked at some unsafe code in action and were introduced to a number of examples before finally looking at how to compile unsafe C# code.



### C# Grammar

In this appendix we are going to take a whirlwind tour of both the lexical and syntactic grammar of the C# language.

##### Lexical Grammar

input::

input-sectionopt input-section::

input-section-part

input-section input-section-part input-section-part::

input-elementsopt new-line pp-directive

input-elements:: input-element

input-elements input-element

input-element:: whitespace comment

token

Comments

comment::

single-line-comment delimited-comment

single-line-comment::

// input-charactersopt input-characters::

input-character

input-characters input-character input-character::

Any Unicode character except a new-line-character new-line-character::

Carriage return character (U+000D) Line feed character (U+000A)

Next line character (U+0085)

Line separator character (U+2028) Paragraph separator character (U+2029)

delimited-comment::

/\* delimited-comment-textopt asterisks /

delimited-comment-text:: delimited-comment-section

delimited-comment-text delimited-comment-section delimited-comment-section::

not-asterisk asterisks not-slash

asterisks::

\*

asterisks \* not-asterisk::

Any Unicode character except \* not-slash::

Any Unicode character except /

Identifiers

identifier::

available-identifier

@ identifier-or-keyword available-identifier::

An identifier-or-keyword that is not a keyword identifier-or-keyword::

identifier-start-character identifier-part-charactersopt identifier-start-character::

letter-character

\_ (the underscore character U+005F)

identifier-part-characters:: identifier-part-character

identifier-part-characters identifier-part-character identifier-part-character::

letter-character decimal-digit-character connecting-character combining-character formatting-character

letter-character::

A Unicode character of classes:

Lu Ll Lt Lm Lo Nl

A unicode-escape-sequence representing a character of classes: Lu

Ll Lt Lm Lo Nl

combining-character::

A Unicode character of classes Mn or Mc

A unicode-escape-sequence representing a character of classes Mn or Mc decimal-digit-character::

A Unicode character of the class Nd

A unicode-escape-sequence representing a character of the class Nd

connecting-character::

A Unicode character of the class Pc

A unicode-escape-sequence representing a character of the class Pc formatting-character::

A Unicode character of the class Cf

A unicode-escape-sequence representing a character of the class Cf

Keywords

keyword:: one of 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 void volatile while

Line Terminators

new-line::

Carriage return character (U+000D) Line feed character (U+000A) Carriage return character (U+000D)

followed by line feed character (U+000A) Next line character (U+2085)

Line separator character (U+2028) Paragraph separator character (U+2029)

Literals

literal::

boolean-literal integer-literal

real-literal character-literal string-literal null-literal

boolean-literal:: true

false

integer-literal::

decimal-integer-literal hexadecimal-integer-literal

decimal-integer-literal::

decimal-digits integer-type-suffixopt decimal-digits::

decimal-digit

decimal-digits decimal-digit decimal-digit:: one of

0 1 2 3 4 5 6 7 8 9

integer-type-suffix:: one of

U u L l UL Ul uL ul LU Lu lU lu hexadecimal-integer-literal::

0x hex-digits integer-type-suffixopt 0X hex-digits integer-type-suffixopt

hex-digits:: hex-digit

hex-digits hex-digit

hex-digit:: one of

0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f real-literal::

decimal-digits . decimal-digits exponent-partopt real-type-suffixopt

. decimal-digits exponent-partopt real-type-suffixopt decimal-digits exponent-part real-type-suffixopt decimal-digits real-type-suffix

exponent-part::

e signopt decimal-digits E signopt decimal-digits

sign:: one of

+ -

real-type-suffix:: one of F f D d M m

character-literal:: ‘ character ‘

character::

single-character simple-escape-sequence

hexadecimal-escape-sequence unicode-escape-sequence

single-character::

Any character except ‘ (U+0027), \ (U+005C), and new-line-character simple-escape-sequence:: one of

\’ \” \\ \0 \a \b \f \n \r \t \v

hexadecimal-escape-sequence::

\x hex-digit hex-digitopt hex-digitopt hex-digitopt string-literal::

regular-string-literal verbatim-string-literal

regular-string-literal::

“ regular-string-literal-charactersopt “ regular-string-literal-characters::

regular-string-literal-character

regular-string-literal-characters regular-string-literal-character regular-string-literal-character::

single-regular-string-literal-character simple-escape-sequence

hexadecimal-escape-sequence unicode-escape-sequence

single-regular-string-literal-character::

Any character except “ (U+0022), \ (U+005C), and new-line-character verbatim-string-literal::

@” verbatim-string-literal-charactersopt “ verbatim-string-literal-characters::

verbatim-string-literal-character

verbatim-string-literal-characters verbatim-string-literal-character verbatim-string-literal-character::

single-verbatim-string-literal-character quote-escape-sequence

single-verbatim-string-literal-character:: Any character except “

quote-escape-sequence:: “”

null-literal:: null

Operators/Punctuators

operator-or-punctuator:: one of

{

}

[

]

(

)

.

,

:

;

+

-

\*

/

%

&

|

^

!

~

=

<

>

?

??

::

++

--

&&

||

->

==

!=

<=

>=

+=

-=

\*=

/=

%=

&=

|=

^=

<<

<<=

right-shift::

> >

right-shift-assignment::

> >=

Pre-Processing Directives

pp-directive:: pp-declaration pp-conditional pp-line

pp-diagnostic pp-region

pp-pragma conditional-symbol::

identifier

Any keyword except true or false pp-expression::

whitespaceopt pp-or-expression whitespaceopt pp-or-expression::

pp-and-expression

pp-or-expression whitespaceopt || whitespaceopt pp-and-expression

pp-and-expression::

pp-equality-expression

pp-and-expression whitespaceopt && whitespaceopt pp-equality-expression pp-equality-expression::

pp-unary-expression

pp-equality-expression whitespaceopt == whitespaceopt pp-unary-expression pp-equality-expression whitespaceopt != whitespaceopt pp-unary-expression

pp-unary-expression:: pp-primary-expression

! whitespaceopt pp-unary-expression pp-primary-expression::

true false

conditional-symbol

( whitespaceopt pp-expression whitespaceopt ) pp-declaration::

whitespaceopt # whitespaceopt define whitespace conditional-symbol pp-new-line whitespaceopt # whitespaceopt undef whitespace conditional-symbol pp-new-line

pp-new-line::

whitespaceopt single-line-commentopt new-line pp-conditional::

pp-if-section pp-elif-sectionsopt pp-else-sectionopt pp-endif pp-if-section::

whitespaceopt # whitespaceopt if whitespace pp-expression pp-new-line conditional-sectionopt

pp-elif-sections:: pp-elif-section

pp-elif-sections pp-elif-section pp-elif-section::

whitespaceopt # whitespaceopt elif whitespace pp-expression pp-new-line conditional-sectionopt

pp-else-section::

whitespaceopt # whitespaceopt else pp-new-line conditional-sectionopt pp-endif::

whitespaceopt # whitespaceopt endif pp-new-line conditional-section::

input-section skipped-section

skipped-section:: skipped-section-part

skipped-section skipped-section-part skipped-section-part::

whitespaceopt skipped-charactersopt new-line pp-directive

skipped-characters::

not-number-sign input-charactersopt not-number-sign::

Any input-character except # pp-line::

whitespaceopt # whitespaceopt line whitespace line-indicator pp-new-line

line-indicator::

decimal-digits whitespace file-name decimal-digits

identifier-or-keyword file-name::

“ file-name-characters “ file-name-characters::

file-name-character

file-name-characters file-name-character file-name-character::

Any character except “ (U+0022), and new-line-character pp-diagnostic::

whitespaceopt # whitespaceopt error pp-message whitespaceopt # whitespaceopt warning pp-message

pp-message:: new-line

whitespace input-charactersopt new-line pp-region::

pp-start-region conditional-sectionopt pp-end-region pp-start-region::

whitespaceopt # whitespaceopt region pp-message pp-end-region::

whitespaceopt # whitespaceopt endregion pp-message pp-pragma:

whitespaceopt # whitespaceopt pragma pp-pragma-text

pp-pragma-text: new-line

whitespace input-charactersopt new-line

Unicode Escape Characters

unicode-escape-sequence::

\u hex-digit hex-digit hex-digit hex-digit

\U hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit

White Space

whitespace::

whitespace-characters whitespace-characters::

whitespace-character

whitespace-characters whitespace-character whitespace-character::

Any character with Unicode class Zs Horizontal tab character (U+0009) Vertical tab character (U+000B) Form feed character (U+000C)

##### Syntactic Grammar

compilation-unit:

extern-alias-directivesopt using-directivesopt global-attributesopt namespace-member-declarationsopt

namespace-name: namespace-or-type-name

type-name:

namespace-or-type-name namespace-or-type-name:

identifier type-argument-listopt qualified-alias-member

namespace-or-type-name . identifier type-argument-listopt

Arrays

array-type:

non-array-type rank-specifiers non-array-type:

value-type class-type interface-type delegate-type type-parameter

rank-specifiers: rank-specifier

rank-specifiers rank-specifier rank-specifier:

[ dim-separatorsopt ] dim-separators:

,

dim-separators ,

array-initializer:

{ variable-initializer-listopt }

{ variable-initializer-list , } variable-initializer-list:

variable-initializer

variable-initializer-list , variable-initializer variable-initializer:

expression

array-initializer

Attributes

global-attributes:

global-attribute-sections global-attribute-sections:

global-attribute-section

global-attribute-sections global-attribute-section global-attribute-section:

[ global-attribute-target-specifier attribute-list ]

[ global-attribute-target-specifier attribute-list , ] global-attribute-target-specifier:

global-attribute-target : global-attribute-target:

identifier keyword

attributes: attribute-sections

attribute-sections: attribute-section

attribute-sections attribute-section attribute-section:

[ attribute-target-specifieropt attribute-list ]

[ attribute-target-specifieropt attribute-list , ] attribute-target-specifier:

attribute-target : attribute-target:

identifier keyword

attribute-list: attribute

attribute-list , attribute attribute:

attribute-name attribute-argumentsopt attribute-name:

type-name attribute-arguments:

( positional-argument-listopt )

( positional-argument-list , named-argument-list ) ( named-argument-list )

positional-argument-list: positional-argument

positional-argument-list , positional-argument

positional-argument:

attribute-argument-expression named-argument-list:

named-argument

named-argument-list , named-argument named-argument:

identifier = attribute-argument-expression attribute-argument-expression:

expression

Classes

class-declaration:

attributesopt class-modifiersopt partialopt class identifier type-parameter-listopt class-baseopt type-parameter-constraints-clausesopt class-body ;opt

class-modifiers: class-modifier

class-modifiers class-modifier class-modifier:

new public protected internal private abstract sealed static

class-base:

: class-type

: interface-type-list

: class-type , interface-type-list interface-type-list:

interface-type

interface-type-list , interface-type class-body:

{ class-member-declarationsopt } class-member-declarations:

class-member-declaration

class-member-declarations class-member-declaration class-member-declaration:

constant-declaration field-declaration method-declaration property-declaration event-declaration indexer-declaration operator-declaration constructor-declaration finalizer-declaration

static-constructor-declaration type-declaration

constant-declaration:

attributesopt constant-modifiersopt const type constant-declarators ; constant-modifiers:

constant-modifier

constant-modifiers constant-modifier constant-modifier:

new public protected internal private

constant-declarators: constant-declarator

constant-declarators , constant-declarator constant-declarator:

identifier = constant-expression field-declaration:

attributesopt field-modifiersopt type variable-declarators ; field-modifiers:

field-modifier

field-modifiers field-modifier field-modifier:

new public protected

internal private static readonly volatile

variable-declarators: variable-declarator

variable-declarators , variable-declarator variable-declarator:

identifier

identifier = variable-initializer variable-initializer:

expression

array-initializer method-declaration:

method-header method-body method-header:

attributesopt method-modifiersopt return-type member-name type-parameter-listopt ( formal-parameter-listopt ) type-parameter-constraints-clausesopt

method-modifiers: method-modifier

method-modifiers method-modifier method-modifier:

new public

protected internal private static virtual sealed override abstract extern

return-type: type

void member-name:

identifier

interface-type . identifier method-body:

block

;

formal-parameter-list: fixed-parameters

fixed-parameters , parameter-array parameter-array

fixed-parameters: fixed-parameter

fixed-parameters , fixed-parameter

fixed-parameter:

attributesopt parameter-modifieropt type identifier parameter-modifier:

ref out

parameter-array:

attributesopt params array-type identifier property-declaration:

attributesopt property-modifiersopt type member-name { accessor-declarations } property-modifiers:

property-modifier

property-modifiers property-modifier property-modifier:

new public protected internal private static virtual sealed override abstract extern

accessor-declarations:

get-accessor-declaration set-accessor-declarationopt set-accessor-declaration get-accessor-declarationopt

get-accessor-declaration:

attributesopt accessor-modifieropt get accessor-body set-accessor-declaration:

attributesopt accessor-modifieropt set accessor-body accessor-modifier:

protected internal private

protected internal internal protected

accessor-body: block

;

event-declaration:

attributesopt event-modifiersopt event type variable-declarators ; attributesopt event-modifiersopt event type member-name

{ event-accessor-declarations } event-modifiers:

event-modifier

event-modifiers event-modifier

event-modifier: new

public protected internal private static virtual sealed override abstract extern

event-accessor-declarations:

add-accessor-declaration remove-accessor-declaration remove-accessor-declaration add-accessor-declaration

add-accessor-declaration: attributesopt add block

remove-accessor-declaration:

attributesopt remove block indexer-declaration:

attributesopt indexer-modifiersopt indexer-declarator { accessor-declarations } indexer-modifiers:

indexer-modifier

indexer-modifiers indexer-modifier

indexer-modifier: new

public protected internal private virtual sealed override abstract extern

indexer-declarator:

type this [ formal-parameter-list ]

type interface-type . this [ formal-parameter-list ] operator-declaration:

attributesopt operator-modifiers operator-declarator operator-body operator-modifiers:

operator-modifier

operator-modifiers operator-modifier operator-modifier:

public static extern

operator-declarator:

unary-operator-declarator binary-operator-declarator conversion-operator-declarator

unary-operator-declarator:

type operator overloadable-unary-operator ( type identifier ) overloadable-unary-operator: one of

+

-

!

~

++

--

true false

binary-operator-declarator:

type operator overloadable-binary-operator ( type identifier , type identifier

)

overloadable-binary-operator: one of

+

-

\*

/

%

&

|

^

<<

right-shift

==

!=

>

<

>=

<=

conversion-operator-declarator:

implicit operator type ( type identifier ) explicit operator type ( type identifier )

operator-body: block

;

constructor-declaration:

attributesopt constructor-modifiersopt constructor-declarator constructor-body constructor-modifiers:

constructor-modifier

constructor-modifiers constructor-modifier constructor-modifier:

public protected

internal private extern

constructor-declarator:

identifier ( formal-parameter-listopt ) constructor-initializeropt constructor-initializer:

: base ( argument-listopt )

: this ( argument-listopt ) constructor-body:

block

;

static-constructor-declaration:

attributesopt static-constructor-modifiers identifier ( ) static-constructor- body

static-constructor-modifiers:

externopt static static externopt

static-constructor-body: block

;

finalizer-declaration:

attributesopt externopt ~ identifier ( ) finalizer-body finalizer-body:

block

;

Delegates

delegate-declaration:

attributesopt delegate-modifiersopt delegate return-type identifier type-parameter-listopt

( formal-parameter-listopt ) type-parameter-constraints-clausesopt ; delegate-modifiers:

delegate-modifier

delegate-modifiers delegate-modifier delegate-modifier:

new public protected internal private

Enums

enum-declaration:

attributesopt enum-modifiersopt enum identifier enum-baseopt enum-body ;opt enum-base:

: integral-type enum-body:

{ enum-member-declarationsopt }

{ enum-member-declarations , } enum-modifiers:

enum-modifier

enum-modifiers enum-modifier

enum-modifier: new

public protected internal private

enum-member-declarations: enum-member-declaration

enum-member-declarations , enum-member-declaration enum-member-declaration:

attributesopt identifier

attributesopt identifier = constant-expression

Expressions

argument-list: argument

argument-list , argument argument:

expression

ref variable-reference out variable-reference

primary-expression:

array-creation-expression

primary-no-array-creation-expression

primary-no-array-creation-expression: literal

simple-name parenthesized-expression member-access

invocation-expression element-access

this-access base-access

post-increment-expression post-decrement-expression object-creation-expression delegate-creation-expression typeof-expression

checked-expression unchecked-expression default-value-expression anonymous-method-expression

simple-name:

identifier type-argument-listopt parenthesized-expression:

( expression ) member-access:

primary-expression . identifier type-argument-listopt predefined-type . identifier type-argument-listopt qualified-alias-member . identifier type-argument-listopt

predefined-type: one of bool

byte char decimal double float int long object sbyte short string uint ulong ushort

invocation-expression:

primary-expression ( argument-listopt ) element-access:

primary-no-array-creation-expression [ expression-list ] expression-list:

expression

expression-list , expression this-access:

this

base-access:

base . identifier type-argument-listopt base [ expression-list ]

post-increment-expression: primary-expression ++

post-decrement-expression: primary-expression --

object-creation-expression: new type ( argument-listopt )

array-creation-expression:

new non-array-type [ expression-list ] rank-specifiersopt array-initializeropt new array-type array-initializer

delegate-creation-expression:

new delegate-type ( expression ) typeof-expression:

typeof ( type )

typeof ( unbound-type-name ) typeof ( void )

unbound-type-name:

identifier generic-dimension-specifieropt

identifier :: identifier generic-dimension-specifieropt unbound-type-name . identifier generic-dimension-specifieropt

generic-dimension-specifier:

< commasopt >

commas:

,

commas ,

checked-expression: checked ( expression )

unchecked-expression: unchecked ( expression )

default-value-expression:

default ( type )

anonymous-method-expression:

delegate anonymous-method-signatureopt block anonymous-method-signature:

( anonymous-method-parameter-listopt ) anonymous-method-parameter-list:

anonymous-method-parameter

anonymous-method-parameter-list , anonymous-method-parameter anonymous-method-parameter:

parameter-modifieropt type identifier unary-expression:

primary-expression

+ unary-expression

- unary-expression

! unary-expression

~ unary-expression

pre-increment-expression pre-decrement-expression cast-expression

pre-increment-expression:

++ unary-expression

pre-decrement-expression:

-- unary-expression cast-expression:

( type ) unary-expression multiplicative-expression:

unary-expression

multiplicative-expression \* unary-expression multiplicative-expression / unary-expression multiplicative-expression % unary-expression

additive-expression: multiplicative-expression

additive-expression + multiplicative-expression

additive-expression – multiplicative-expressionshift-expression: additive-expression

shift-expression << additive-expression

shift-expression right-shift additive-expression relational-expression:

shift-expression

relational-expression < shift-expression relational-expression > shift-expression

relational-expression <= shift-expression relational-expression >= shift-expression relational-expression is type

relational-expression as type equality-expression:

relational-expression

equality-expression == relational-expression equality-expression != relational-expression

and-expression: equality-expression

and-expression & equality-expression exclusive-or-expression:

and-expression

exclusive-or-expression ^ and-expression inclusive-or-expression:

exclusive-or-expression

inclusive-or-expression | exclusive-or-expression conditional-and-expression:

inclusive-or-expression

conditional-and-expression && inclusive-or-expression conditional-or-expression:

conditional-and-expression

conditional-or-expression || conditional-and-expression

null-coalescing-expression: conditional-or-expression

conditional-or-expression ?? null-coalescing-expression conditional-expression:

null-coalescing-expression

null-coalescing-expression ? expression : expression assignment:

unary-expression assignment-operator expression assignment-operator: one of

=

+=

-=

\*=

/=

%=

&=

|=

^=

<<=

right-shift-assignment expression:

conditional-expression assignment

constant-expression: expression

boolean-expression: expression

Generics

type-parameter-list:

< type-parameters > type-parameters:

attributesopt type-parameter

type-parameters , attributesopt type-parameter type-parameter:

identifier

type-argument-list:

< type-arguments > type-arguments:

type-argument

type-arguments , type-argument type-argument:

type

type-parameter-constraints-clauses: type-parameter-constraints-clause

type-parameter-constraints-clauses type-parameter-constraints-clause type-parameter-constraints-clause:

where type-parameter : type-parameter-constraints type-parameter-constraints:

primary-constraint secondary-constraints

constructor-constraint

primary-constraint , secondary-constraints primary-constraint , constructor-constraint secondary-constraints , constructor-constraint

primary-constraint , secondary-constraints , constructor-constraint primary-constraint:

class-type class struct

secondary-constraints: interface-type

type-parameter

secondary-constraints , interface-type secondary-constraints , type-parameter

constructor-constraint: new ( )

Interfaces

interface-declaration:

attributesopt interface-modifiersopt partialopt interface identifier type-parameter-listopt

interface-baseopt type-parameter-constraints-clausesopt interface-body ;opt interface-modifiers:

interface-modifier

interface-modifiers interface-modifier

interface-modifier: new

public protected internal private

interface-base:

: interface-type-list interface-body:

{ interface-member-declarationsopt } interface-member-declarations:

interface-member-declaration

interface-member-declarations interface-member-declaration interface-member-declaration:

interface-method-declaration interface-property-declaration interface-event-declaration interface-indexer-declaration

interface-method-declaration:

attributesopt newopt return-type identifier type-parameter-listopt

( formal-parameter-listopt ) type-parameter-constraints-clausesopt ; interface-property-declaration:

attributesopt newopt type identifier { interface-accessors }

interface-accessors: attributesopt get ;

attributesopt set ;

attributesopt get ; attributesopt set ; attributesopt set ; attributesopt get ;

interface-event-declaration:

attributesopt newopt event type identifier ; interface-indexer-declaration:

attributesopt newopt type this [ formal-parameter-list ] { interface-accessors }

Statements

statement:

labeled-statement declaration-statement embedded-statement

embedded-statement: block

empty-statement expression-statement selection-statement iteration-statement jump-statement

try-statement checked-statement unchecked-statement lock-statement

using-statement yield-statement

block:

{ statement-listopt } statement-list:

statement

statement-list statement empty-statement:

;

labeled-statement: identifier : statement

declaration-statement:

local-variable-declaration ; local-constant-declaration ;

local-variable-declaration:

type local-variable-declarators local-variable-declarators:

local-variable-declarator

local-variable-declarators , local-variable-declarator local-variable-declarator:

identifier

identifier = local-variable-initializer local-variable-initializer:

expression

array-initializer

local-constant-declaration:

const type constant-declarators constant-declarators:

constant-declarator

constant-declarators , constant-declarator constant-declarator:

identifier = constant-expression expression-statement:

statement-expression ; statement-expression:

invocation-expression object-creation-expression assignment

post-increment-expression post-decrement-expression pre-increment-expression pre-decrement-expression

selection-statement: if-statement switch-statement

if-statement:

if ( boolean-expression ) embedded-statement

if ( boolean-expression ) embedded-statement else embedded-statement switch-statement:

switch ( expression ) switch-block

switch-block:

{ switch-sectionsopt } switch-sections:

switch-section

switch-sections switch-section switch-section:

switch-labels statement-list switch-labels:

switch-label

switch-labels switch-label switch-label:

case constant-expression :

default : iteration-statement:

while-statement do-statement for-statement

foreach-statement while-statement:

while ( boolean-expression ) embedded-statement do-statement:

do embedded-statement while ( boolean-expression ) ; for-statement:

for ( for-initializeropt ; for-conditionopt ; for-iteratoropt ) embedded-statement

for-initializer:

local-variable-declaration statement-expression-list

for-condition: boolean-expression

for-iterator:

statement-expression-list statement-expression-list:

statement-expression

statement-expression-list , statement-expression foreach-statement:

foreach ( type identifier in expression ) embedded-statement jump-statement:

break-statement continue-statement goto-statement return-statement throw-statement

break-statement: break ;

continue-statement: continue ;

goto-statement: goto identifier ;

goto case constant-expression ; goto default ;

return-statement: return expressionopt ;

throw-statement:

throw expressionopt ; try-statement:

try block catch-clauses

try block catch-clausesopt finally-clause catch-clauses:

specific-catch-clauses

specific-catch-clausesopt general-catch-clause specific-catch-clauses:

specific-catch-clause

specific-catch-clauses specific-catch-clause specific-catch-clause:

catch ( class-type identifieropt ) block general-catch-clause:

catch block finally-clause:

finally block checked-statement:

checked block unchecked-statement:

unchecked block

lock-statement:

lock ( expression ) embedded-statement using-statement:

using ( resource-acquisition ) embedded-statement resource-acquisition:

local-variable-declaration expression

yield-statement:

yield return expression ; yield break ;

namespace-declaration:

namespace qualified-identifier namespace-body ;opt qualified-identifier:

identifier

qualified-identifier . identifier namespace-body:

{ extern-alias-directivesopt using-directivesopt namespace-member-declarationsopt

}

extern-alias-directives: extern-alias-directive

extern-alias-directives extern-alias-directive extern-alias-directive:

extern alias identifier ; using-directives:

using-directive

using-directives using-directive

using-directive:

using-alias-directive using-namespace-directive

using-alias-directive:

using identifier = namespace-or-type-name ; using-namespace-directive:

using namespace-name ; namespace-member-declarations:

namespace-member-declaration

namespace-member-declarations namespace-member-declaration namespace-member-declaration:

namespace-declaration type-declaration

type-declaration: class-declaration struct-declaration

interface-declaration enum-declaration delegate-declaration

qualified-alias-member:

identifier :: identifier type-argument-listopt

Structs

struct-declaration:

attributesopt struct-modifiersopt partialopt struct identifier type-parameter-listopt

struct-interfacesopt type-parameter-constraints-clausesopt struct-body ;opt

struct-modifiers: struct-modifier

struct-modifiers struct-modifier struct-modifier:

new public protected internal private

struct-interfaces:

: interface-type-list struct-body:

{ struct-member-declarationsopt } struct-member-declarations:

struct-member-declaration

struct-member-declarations struct-member-declaration struct-member-declaration:

constant-declaration field-declaration method-declaration property-declaration event-declaration indexer-declaration operator-declaration constructor-declaration

static-constructor-declaration type-declaration

Types

type:

value-type reference-type type-parameter value-type: struct-type enum-type

struct-type: type-name simple-type nullable-type

simple-type: numeric-type bool

numeric-type: integral-type floating-point-type decimal

integral-type: sbyte

byte short

ushort int uint long ulong char

floating-point-type: float

double enum-type:

type-name nullable-type:

non-nullable-value-type ? non-nullable-value-type: enum-type

type-name simple-type

reference-type: class-type interface-type array-type delegate-type

class-type: type-name object string

interface-type: type-name

array-type:

non-array-type rank-specifiers non-array-type:

value-type class-type interface-type delegate-type type-parameter

rank-specifiers: rank-specifier

rank-specifiers rank-specifier rank-specifier:

[ dim-separatorsopt ] dim-separators:

,

dim-separators , delegate-type:

type-name

Variables

variable-reference: expression

##### Extensions for Unsafe Code

class-modifier:

...

unsafe

struct-modifier:

...

unsafe

interface-modifier:

...

unsafe

delegate-modifier:

...

unsafe

field-modifier:

...

unsafe

method-modifier:

...

unsafe

property-modifier:

...

unsafe

event-modifier:

...

unsafe

indexer-modifier:

...

unsafe

operator-modifier:

...

unsafe

constructor-modifier:

...

unsafe

finalizer-declaration:

attributesopt externopt unsafeopt ~ identifier ( ) finalizer-body attributesopt unsafeopt externopt ~ identifier ( ) finalizer-body

static-constructor-modifiers: externopt unsafeopt static

unsafeopt externopt static

externopt static unsafeopt unsafeopt static externopt static externopt unsafeopt static unsafeopt externopt

embedded-statement:

...

unsafe-statement unsafe-statement:

unsafe block

type:

value-type reference-type type-parameter pointer-type

pointer-type: unmanaged-type \* void \*

unmanaged-type: type

primary-no-array-creation-expression:

...

sizeof-expression

primary-no-array-creation-expression:

...

pointer-member-access pointer-element-access

unary-expression:

...

pointer-indirection-expression addressof-expression

pointer-indirection-expression:

\* unary-expression pointer-member-access:

primary-expression -> identifier type-argument-listopt

pointer-element-access:

primary-no-array-creation-expression [ expression ] addressof-expression:

& unary-expression sizeof-expression:

sizeof ( unmanaged-type ) embedded-statement:

...

fixed-statement fixed-statement:

fixed ( pointer-type fixed-pointer-declarators ) embedded-statement fixed-pointer-declarators:

fixed-pointer-declarator

fixed-pointer-declarators , fixed-pointer-declarator fixed-pointer-declarator:

identifier = fixed-pointer-initializer fixed-pointer-initializer:

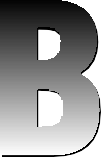
& variable-reference expression

local-variable-initializer: expression

array-initializer stackalloc-initializer

stackalloc-initializer:

stackalloc unmanaged-type [ expression ]



### Naming Conventions

Consistent naming is important in coding because it adds to the level of predictability and discov- erability in managed class libraries. The more you adopt a standardized naming convention, the easier the code is to read and follow and the fewer issues you should encounter. For the hobbyist this means fewer problems; for the professional this means that they can get more done in less time and that saves money.

This appendix provides a naming convention for .NET Framework types. For each type, attention should be paid to capitalization, case, and word choice.

##### Capitalization

There are three conventions to use for naming identifiers:

* Pascal Case
* Camel case
* Uppercase

Pascal Case

The first letter in the identifier and then the first letter of each subsequent concatenated word are capitalized (with no spaces added).

Use Pascal case for identifiers of three or more characters.

ButtonStyle

Camel Case

The first letter of an identifier is lowercase, and then the first letter of each subsequent concatenated word is capitalized (with no spaces added).

buttonStyle

Uppercase

All the letters in the identifier are capitalized. Use this for identifiers that consist of two or fewer letters.

System.IO

In the following table, the capitalization rules are summarised for different identifiers.

**Identifier Case**

Public instance field Pascal

Class

Enum type Enum value Event Excepion class

Read-only Static field Interface

Method Namespace Parameter Property

Protected instance field

Pascal

Pascal Pascal Pascal Pascal Pascal Pascal Pascal Pascal Camel Pascal Camel

##### Case Sensitivity

The following are the rules for case sensitivity and help to ensure cross-language interoperability:

❑ Do not use names that require case sensitivity.

❑ Do not create two or more namespaces that differ by case alone.

❑ Do not create a function with a parameter name that differs only in the case of the parameter.

❑ Do not create a namespace with type names that differ only by case.

❑ Do not create a type with property names that differ only by case.

❑ Do not create a type with method names that differ only by case.

##### Abbreviations

The following are the rules for case sensitivity and help to ensure cross-language interoperability.

❑ Do not use abbreviations or contractions as parts of identifier names.

❑ Do not use obscure acronyms.

❑ Use well-known acronyms to replace long phrases.

❑ Use the appropriate case rules for acronyms (Pascal, camel, and uppercase).

❑ Do not use abbreviations in identifiers or parameter names. If abbreviations must be used, always use camel case.

##### Keywords to Avoid

Avoid using any class names that duplicate commonly used .NET Framework namespaces.

Also, avoid using identifiers that conflict with the following keywords listed in the following table.

|  |  |  |
| --- | --- | --- |
| AddHandler | AddressOf | Alias |
| And | Ansi | As |
| Assembly | Auto | Base |
| Boolean | ByRef | Byte |
| ByVal | Call | Case |
| Catch | CBool | CByte |
| CChar | CDate | Cdec |
| CDbl | Char | CInt |
| Class | CLng | CObj |
| Const | CShort | CSng |
| CStr | CType | Date |

|  |  |  |
| --- | --- | --- |
| Decimal | Declare | Default |
| Delegate | Dim | Do |
| Double | Each | Else |
| ElseIf | End | Enum |
| Erase | Error | Event |
| Exit | ExternalSource | False |
| Finalize | Finally | Float |
| For | Friend | Function |
| Get | GetType | Goto |
| Handles | If | Implements |
| Imports | In | Inherits |
| Integer | Interface | Is |
| Let | Lib | Like |
| Long | Loop | Me |
| Mod | Module | MustInherit |
| MustOverride | MyBase | MyClass |
| Namespace | New | Next |
| Not | Nothing | NotInheritable |
| NotOverridable | Object | On |
| Option | Optional | Or |
| Overloads | Overridable | Overrides |
| ParamArray | Preserve | Private |
| Property | Protected | Public |
| RaiseEvent | ReadOnly | ReDim |
| Region | REM | RemoveHandler |

|  |  |  |
| --- | --- | --- |
| Resume | Return | Select |
| Set | Shadows | Shared |
| Short | Single | Static |
| Step | Stop | String |
| Structure | Sub | SyncLock |
| Then | Throw | To |
| True | Try | TypeOf |
| Unicode | Until | volatile |
| When | While | With |
| WithEvents | WriteOnly | Xor |
| eval | Extends | instanceof |
| package | Var |  |

##### Namespace Naming

As a rule, namespace names should be composed of the company name followed by the technology name and then optionally the feature and design.

CompanyName.TechnologyName[.Feature][.Design]

Always use Pascal class for naming and separate logical components with periods. Use plurals where appropriate.

Do not use the same name for namespace and class.

##### Class Naming

Use the following rules for naming classes:

❑ Use a noun (or noun phrase) to name a class.

❑ Use Pascal case.

❑ Use abbreviations sparingly and with care to avoid confusion.

❑ Do not use type prefixes as class names.

❑ Do not use the underscore character (\_).

❑ Use compound words to name a derived class where appropriate.

❑ At times it might be necessary to have class names that begin with the letter I even when the class is not itself an interface (that is, the class has a name beginning with the letter I).

##### Interface Naming

Use the following rules for naming interfaces:

❑ Use a noun (or noun phrase) or an adjective that describes behavior.

❑ Use Pascal case.

❑ Use abbreviations sparingly and with care to avoid confusion.

❑ Do not use the underscore character (\_).

❑ Prefix interfaces with the letter I.

##### Attribute Naming

Use the following rules for naming attributes:

❑ Always add the suffix Attribute to custom attribute classes.

##### Enumeration Type Naming

Use the following rules for naming enumerations:

❑ Use Pascal case.

❑ Use abbreviations sparingly and with care to avoid confusion.

❑ Do not use the Enum suffix on Enum type names.

❑ Use singular names except for Enum types that are bit fields.

❑ Always add the FlagAttributes to a bit field Enum type.

##### Static Field Naming

Use the following rules for naming static fields:

❑ Use Pascal case.

❑ Do not use Hungarian Notation (this is a common notation style, but it is not recommended for

.NET programming).

❑ Use nouns (or noun phrases) or abbreviation of nouns.

##### Parameter Naming

Use the following rules for naming parameters:

❑ Use camel case.

❑ Use descriptive parameter names.

❑ Do not use reserved parameters.

❑ Do not prefix with Hungarian Notation (again, this notation is not recommended for .NET programming).

##### Method Naming

Use the following rules for naming methods:

❑ Use Pascal case.

❑ Use verbs (or verb phrases).

##### Property Naming

Use the following rules for naming properties:

❑ Use a noun (or noun phrase).

❑ Use Pascal case.

❑ Do not use Hungarian Notation.

❑ Consider creating a property that has the same name as the underlying type.

##### Event Naming

Use the following rules for naming events:

❑ Do not use Hungarian Notation.

❑ Use Pascal case.

❑ Use the EventHandler suffix on event handler names.

❑ Specify two parameters — sender, which represents the object that raised the event, and e, which is the state associated with the event encapsulated in an instance on an event class.

❑ Give event argument classes the EventArgs suffix.

❑ Name events with a verb where possible.

❑ Do not use a prefix or suffix on the event declaration.



### Standard Library

A conforming C# implementation has to provide a minimum set of types that have a specific semantic. These types, along with their corresponding members, are listed below.

All type names that start with System are for the use of the standard library. Those currently not in use might be used in the future.

The standard library is the minimum set of types and members required by conforming to a C# implementation. This listing contains only the members required by the C# language.

This is not a complete listing; any C# implementation will supply a much more comprehensive library. For example:

❑ Adding namespaces

❑ Adding types

❑ Adding members to noninterface types

❑ Struct and class types implementing additional interfaces

❑ Adding more attributes to types and members

❑ The following is included for reference. For the full text, refer to the ECMA 334 C# lan- guage specification:

namespace System

{

public class ApplicationException : Exception

{

public ApplicationException();

public ApplicationException(string message);

public ApplicationException(string message, Exception innerException);

}

}

public class ArgumentException : SystemException

{

public ArgumentException();

public ArgumentException(string message);

public ArgumentException(string message, Exception innerException);

}

}

namespace System

{

public class ArithmeticException : SystemException

{

public ArithmeticException();

public ArithmeticException(string message);

public ArithmeticException(string message, Exception innerException);

}

}

namespace System

{

public abstract class Array : IList, ICollection, IEnumerable

{

public int Length { get; } public int Rank { get; }

public int GetLength(int dimension);

}

}

namespace System

{

public class ArrayTypeMismatchException : SystemException

{

public ArrayTypeMismatchException();

public ArrayTypeMismatchException(string message); public ArrayTypeMismatchException(string message,

Exception innerException);

}

}

namespace System

{

[AttributeUsageAttribute(AttributeTargets.All, Inherited = true, AllowMultiple = false)]

public abstract class Attribute

{

protected Attribute();

}

}

namespace System

{

public enum AttributeTargets

{

Assembly = 1,

Module = 2,

Class = 4,

Struct = 8,

Enum = 16,

Constructor = 32,

Method = 64,

Property = 128,

Field = 256,

Event = 512,

Interface = 1024,

Parameter = 2048,

Delegate = 4096,

ReturnValue = 8192,

GenericParameter = 16384,

All = 32767

}

}

namespace System

{

[AttributeUsageAttribute(AttributeTargets.Class, Inherited = true)] public sealed class AttributeUsageAttribute : Attribute

{

public AttributeUsageAttribute(AttributeTargets validOn); public bool AllowMultiple { get; set; }

public bool Inherited { get; set; } public AttributeTargets ValidOn { get; }

}

}

public struct Boolean

{

}

}

namespace System

{

public struct Byte

{

}

}

namespace System

{

public struct Char

{

}

}

namespace System

{

public struct Decimal

{

}

}

namespace System

{

public abstract class Delegate

{

}

}

namespace System

{

public class DivideByZeroException : ArithmeticException

{

public DivideByZeroException();

public DivideByZeroException(string message);

public DivideByZeroException(string message, Exception innerException);

}

}

namespace System

{

public struct Double

{

}

}

namespace System

{

public abstract class Enum : ValueType

{

protected Enum();

}

}

namespace System

{

public class Exception

{

public Exception();

public Exception(string message);

public Exception(string message, Exception innerException); public sealed Exception InnerException { get; }

public virtual string Message { get; }

}

}

namespace System

{

public interface IDisposable

{

public void Dispose();

}

}

public sealed class IndexOutOfRangeException : SystemException

{

public IndexOutOfRangeException();

public IndexOutOfRangeException(string message); public IndexOutOfRangeException(string message, Exception innerException);

}

}

namespace System

{

public struct Int16

{

}

}

namespace System

{

public struct Int32

{

}

}

namespace System

{

public struct Int64

{

}

}

namespace System

{

public class InvalidCastException : SystemException

{

public InvalidCastException();

public InvalidCastException(string message);

public InvalidCastException(string message, Exception innerException);

}

}

namespace System

{

public class InvalidOperationException : SystemException

{

public InvalidOperationException();

public InvalidOperationException(string message); public InvalidOperationException(string message, Exception innerException);

}

}

namespace System

{

public abstract class MemberInfo

{

protected MemberInfo();

}

}

namespace System

{

public class NotSupportedException : SystemException

{

public NotSupportedException();

public NotSupportedException(string message);

public NotSupportedException(string message, Exception innerException);

}

}

namespace System

{

public struct Nullable<T>

{

public bool HasValue { get; } public T Value { get; }

}

}

public class NullReferenceException : SystemException

{

public NullReferenceException();

public NullReferenceException(string message);

public NullReferenceException(string message, Exception innerException);

}

}

namespace System

{

public class Object

{

public Object();

~Object();

public virtual bool Equals(object obj); public virtual int GetHashCode(); public Type GetType();

public virtual string ToString();

}

}

namespace System

{

[AttributeUsageAttribute(AttributeTargets.Class

| AttributeTargets.Struct

| AttributeTargets.Enum | AttributeTargets.Interface

| AttributeTargets.Constructor | AttributeTargets.Method

| AttributeTargets.Property | AttributeTargets.Field

| AttributeTargets.Event | AttributeTargets.Delegate, Inherited = false)]

public sealed class ObsoleteAttribute : Attribute

{

public ObsoleteAttribute();

public ObsoleteAttribute(string message);

public ObsoleteAttribute(string message, bool error); public bool IsError { get; }

public string Message { get; }

}

}

namespace System

{

public class OutOfMemoryException : SystemException

{

public OutOfMemoryException();

public OutOfMemoryException(string message);

public OutOfMemoryException(string message, Exception innerException);

}

}

namespace System

{

public class OverflowException : ArithmeticException

{

public OverflowException();

public OverflowException(string message);

public OverflowException(string message, Exception innerException);

}

}

namespace System

{

public struct SByte

{

}

}

namespace System

{

public struct Single

{

}

}

namespace System

{

public sealed class StackOverflowException : SystemException

{

public StackOverflowException();

public StackOverflowException(string message);

public StackOverflowException(string message, Exception innerException);

}

}

public sealed class String : IEnumerable<Char>, IEnumerable

{

public int Length { get; }

public char this[int index] { get; }

}

}

namespace System

{

public class SystemException : Exception

{

public SystemException();

public SystemException(string message);

public SystemException(string message, Exception innerException);

}

}

namespace System

{

public abstract class Type : MemberInfo

{

}

}

namespace System

{

public sealed class TypeInitializationException : SystemException

{

public TypeInitializationException(string fullTypeName, Exception innerException);

}

}

namespace System

{

public struct UInt16

{

}

}

namespace System

{

public struct UInt32

{

}

}

namespace System

{

public struct UInt64

{

}

}

namespace System

{

public abstract class ValueType

{

protected ValueType();

}

}

namespace System.Collections

{

public interface ICollection : IEnumerable

{

public int Count { get; }

public bool IsSynchronized { get; } public object SyncRoot { get; }

public void CopyTo(Array array, int index);

}

}

namespace System.Collections

{

public interface IEnumerable

{

public IEnumerator GetEnumerator();

}

}

namespace System.Collections

{

public interface IEnumerator

{

public object Current { get; } public bool MoveNext();

public void Reset();

}

}

namespace System.Collections

{

public interface IList : ICollection, IEnumerable

{

public bool IsFixedSize { get; } public bool IsReadOnly { get; }

public object this[int index] { get; set; } public int Add(object value);

public void Clear();

public bool Contains(object value); public int IndexOf(object value);

public void Insert(int index, object value); public void Remove(object value);

public void RemoveAt(int index);

}

}

namespace System.Collections.Generic

{

public interface ICollection<T> : IEnumerable<T>

{

public int Count { get; } public bool IsReadOnly { get; } public void Add(T item);

public void Clear();

public bool Contains(T item);

public void CopyTo(T[] array, int arrayIndex); public bool Remove(T item);

}

}

namespace System.Collections.Generic

{

public interface IEnumerable<T> : IEnumerable

{

public IEnumerator<T> GetEnumerator();

}

}

namespace System.Collections.Generic

{

public interface IEnumerator<T> : IDisposable, IEnumerator

{

public T Current { get; }

}

}

namespace System.Collections.Generic

{

public interface IList<T> : ICollection<T>

{

public T this[int index] { get; set; } public int IndexOf(T item);

public void Insert(int index, T item); public void RemoveAt(int index);

}

}

namespace System.Diagnostics

{

[AttributeUsageAttribute(AttributeTargets.Method

| AttributeTargets.Class, AllowMultiple = true)] public sealed class ConditionalAttribute : Attribute

{

public ConditionalAttribute(string conditionString); public string ConditionString { get; }

}

}

namespace System.Threading

{

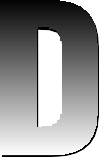
public static class Monitor

{

public static void Enter(object obj); public static void Exit(object obj);

}

}



### Portability

This appendix covers portability issues with C# programs.

##### General Portability Issues

One of the biggest benefits in terms of portability that C# offers is how it leverages the Common Language Runtime (CLR). When .NET programs are compiled, the source code produces both metadata and Microsoft Intermediate Language (MSIL) code. The metadata contains a complete specification for the program, including all the types. Not included are the implementations of the functions though. The CLR then uses this information to activate a .NET program at runtime.

Platform/OS Portability

Because of this reliance on the CLR at runtime, programs can be run without the need for recompi- lation on any operating system or processor (or combinations thereof) that supports the Common Language Runtime. This is because the CLR’s Just-In-Time (JIT) compiler compiles the MSIL code into native code that can be run on the platform.

Simplified Deployment

The assembly produced is a completely self-describing package. This package contains all the metadata and MSIL for the program in question. This means that deployment is as easy as copying the assembly to the desired PC.

Interoperability with Legacy Code

The Common Type System (CTS) that is part of the CLR defines the types that can be expressed in both the metadata and the MSIL, along with the operations that can be carried out on these types. The CTS supports a variety of different languages, both Microsoft and third-party. For example:

❑ C#

❑ Visual Basic .NET

❑ Visual C++ .NET

❑ COBOL

❑ Eiffel

❑ Mercury

❑ ML

❑ Pearl

❑ Python

❑ Smalltalk

The Microsoft CLR makes interoperability with a wide range of existing software written in COM and C easy. The CLR provides PInvoke, a mechanism that enables C functions, structs, and callbacks to be used from within .NET programs. .NET types can also be exposed as COM types, and COM types can be imported as .NET types.

##### Undefined Behavior

A program that does not contain an occurrence of the unsafe modifier cannot exhibit any undefined behavior.

A behavior is undefined as follows:

❑ The initial content of memory when allocated by stackalloc

❑ When attempting to allocate a negative number of items using stackalloc

❑ When trying to dereference the result of converting one pointer type to another when the result- ing pointer is not correctly aligned for the pointer-to type

❑ When applying the unary operator (\*) to a pointer containing an invalid value

❑ When subscripting a pointer to access an out-of-bounds element

❑ Modifying the objects of a managed type using fixed pointers

##### Implementation-Defined Behavior

A conforming implementation is required to document the choice of behavior in each of the areas listed below.

The following are all implementation-defined:

❑ The purpose of a *line-indicator* with an *identifier-or-keyword* whose value does not equal default

❑ The interpretation of the *input-characters* in the *pp-pragma-text* of any #pragma directive

❑ The value of any application parameter passed to main by the host environment before the application has started

Portability

❑ When a System.ArithmeticException (or a subclass) is thrown or an overflow goes unreported when the resulting value is a left operand

❑ When in an unchecked context and the left operand of the division on any integer is the maxi- mum negative int or long value and the right operand is set to -1

❑ When a System.ArithmeticException (or a subclass) is thrown during a decimal remainder operation

❑ Linkage to an external function

❑ Thread termination when there is no matching catch clause and the code that started the thread is reached

❑ The purpose of any attribute target specifies other then those defined by the standard

❑ The mapping between any pointers and integers

❑ The effect of applying a unary operator (\*) to a null pointer

❑ Any behavior when the pointer arithmetic overflows the domain of the pointer type

❑ The result of the sizeof operator for any non pre-defined value types

❑ Any behavior of the fixed statement if the array expression is null or if the array contains zero elements

❑ Any behavior of a fixed statement if the string expression is null

❑ The value returned when a stack allocation of zero size is made

##### Unspecified Behavior

The following is considered unspecified behavior:

❑ The time at which the finalizer for an object is run (once the object has become eligible for finalization)

❑ The value of a result when converting out-of-range values from float or double values to an integral type in an unchecked context

❑ The layout of arrays (except in an unsafe context)

❑ Whether there is any way to execute the *block* on an autonomous method other than through the evaluation and invocation of the *autonomous-method-expression*

❑ The invocation list of a delegate produced from the *autonomous-method-expression* which contains a single entry. The exact target object and target methods of the delegate are unspecified.

❑ The exact timing of static field initializations

❑ The behavior of any uncaught exceptions that occur during finalizer execution

❑ The attributes of a type declared in multiple parts will be determined by combining the attributes of each part in an unspecified order.

❑ The order in which members are placed into a struct

361

* When an enumerator object is in the *running* state, the result of invoking MoveNext is unspecified.
* When an enumerator object is in the *before*, *running* or *after* states, the result of invoking

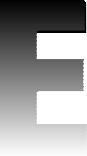
Current is unspecified.

* When an enumerator object is in the *running* state, the result of invoking Dispose is unspecified.

##### Miscellaneous Issues

Here are a few final issues:

* The precise results of floating-point expression evaluations can vary from one implementation to another. This is because different implementations are allowed to evaluate floating-point with varying degrees of precision.
* The CLI (Common Language Infrastructure) reserves certain signatures to maintain cross- compatibility with other programming languages.



### XML Documentation Comments

There is a mechanism in C# that allows programmers to document their code using a specific com- ment syntax that contains XML. These comments are called documentation comments, and the tool used to generate the XML is called the documentation generator, which may or may not be the com- piler used to compile the C# source code. The resulting output is called the documentation file, and any viewer used to display the information contained in this file is called the documentation viewer.

It is important to note that a C# compiler (even one conforming to the specification) does not have to check the syntax of the documentation comments for validity (much in the same way that a compiler doesn’t check the syntax of any other comment you put in the source code). However,

it is perfectly acceptable for a conforming compiler to do this.

##### Syntax

XML documentation comments can be added to the source code by using special single line and delimited comment tags, as shown in the following code lines:

/// single line document comment

/\*\* multi-line delimited document comment \*/

These comments need to immediately precede a user-defined type (for example, a class, delegate, or interface) or a member (for example, event, property, or method) that they are annotating. Since attribute selections are part of the declarations, document comments must come before attributes applied to a type of member.

When using single line comments, if there is a whitespace character following the ///, this will not be included in the XML output. This means that both:

/// Document comment goes here

and

Appendix E

///Document comment goes here

return the same output.

When using delimited document comments, if the first nonwhitespace character on the second line is an asterisk and the same pattern of optional whitespace characters and asterisk characters is repeated at the beginning of each line within the delimited comments, these are not included in the output.

Note that this repeated pattern can include whitespace characters both before and after the asterisk character.

The following shows a valid comment block for code written in C#:

/\*\*

\*

\*

\*

\*

\*

\*

\*/

Comments can be included anywhere inside the block, for example:

/\*\* Comments

* comments
* comments
* comments
* comments
* comments
* comments

\*/

However, note that the following is invalid:

/\*\*

\*

\*

\*

\*

\*

\*

\*/ Comments can’t go here!

All XML documentation comments must be well formed, as laid out in the XL rules at the W3C ([http://www.w3.org/TR/REC-xml).](http://www.w3.org/TR/REC-xml))

Although developers are free to create their own tags for marking up the documentation, a few recom- mended tags have a special meaning.

* <param> — This tag is used to describe parameters. If this tag is used, the document generator must verify that the specified parameters exist. It must also check to see if all of the parameters are described in the documentation. If the checks fail, a warning should be issued.

XML Documentation Comments

* + cref — This attribute is attached to tags that provide a reference to code elements. Code elements that contain code that makes use of generics cannot make use of the generic syntax. For example:

List<T>

The preceding would be invalid and curly braces would need to be used:

List{T}

or the XML escape syntax:

List&lt;T&gt;

* + <summary> — This is intended for use by the documentation viewer to display additional infor- mation about types or members.

Recommended Tags

The following table lists tags that provide commonly used functionality in user documentation.

**Tag Purpose**

<c>

<code>

<example>

<exception>

<list>

<para>

<param>

<paramref>

<permission>

<remark>

<returns>

<see>

<seealso>

<summary>

<typeparam>

<typeparamref>

<value>

Sets text in a code-like proportional font

Sets one or more lines of source code or program output in a code-like proportional font

Indicates an example Identifies an exception Creates a list or table

Allows structure to be added to text output Describes a parameter

Identifies a word that is a parameter name Documents the security accessibility of a member Describes a type

Describes the return value of a method Specifies a link

Generates a See Also entry

Describes the member of a particular type

Describes a type parameter for a method or generic type Identifies a word that is a type parameter name Describes a property

365

## Index

### Index

Index

#### SYMBOLS

&& **(double ampersand) expression, 96–97**

|| **(double pipe) expression, 97**

! **(exclamation point) expression, 97**

? **(question mark) expressions, rules for, 98**

#### A

**abbreviations, naming conventions for, 339**

abstract **accessor**

events, 185

properties, 183

**abstract (keyword), 279**

**abstract methods, 181**

**access modifiers, 174**

**accessibility, 28**

**accessors, 182–185**

**additive operators, 25, 114**

**advantages of C#, 1–2, 5**

**alias directives, 250 alternative text editors, 15 ambiguities in grammar, 37–38 and operators, 112–119**

**anonymous method conversions, 109**

**anonymous methods, 131 Antechinus C# Editor, 15 application startup, 57–58**

**application termination, 58**

**argument lists, 124**

**arithmetic operators, 131–132**

**array covariance, 205–206**

**array types, 22–23, 79, 203–204 arrays**

access, 128

accessing array elements, 205 creating, 205

elements, 85–86

initializers, 206–208

jagged arrays, 22–23

members, 62, 205

overview, 201–203

rectangular arrays, 22–23

syntactic grammar, 293–294

System.Array type, 204

trailing commas, 208

**as (keyword), 279**

**assignment operators, 26, 115, 135**

**assignments, 198**

attributes

**attributes**

compilation, 237

Conditional attribute, 238–239

global attributes, 36

instances, 236

named parameters, 232

naming, 342

Obsolete attribute, 239–240

overview, 231–232

parameter types, 233

positional parameters, 232

reserved attributes, 238

runtime retrieval of attribute instances, 237 specification, 233–236

syntactic grammar, 294–296

usage of, 232–233

#### B

**base (keyword), 279**

**base types, 120**

base-access**, 128–129**

base-class **specification, 171 binary operator**

overload resolution, 118

overview, 113, 189

block **statements, rules for, 91 body**

interfaces, 212

structs, 195

**bool (keyword), 279**

bool **type, 20, 77**

**Boolean expressions, 138**

**Boolean literals, 47**

**boxing conversion, 80, 102, 198**

**break (keyword), 279**

break **statement**

overview, 156–157

rules for, 93

**byte (keyword), 279**

byte **type, 20, 74–75**

**C**

**calling generic methods, 251–252 Camel case, 338**

**capitalization, 337**

**case (keyword), 279**

**case sensitivity, 338–339**

**cast expressions, 131**

**catch clause, not finding, 229 catch (keyword), 279**

**char (keyword), 279**

char **type, 20, 74–75**

**character literals, 48–49**

**checked (keyword), 279**

checked **operator, 130**

checked **statements, rules for, 91 circular references, avoiding, 219 class (keyword), 279**

**class type, 79**

class-base **specification, 171**

class-body**, 171**

**classes. *See also specific classes***

access modifiers, 174

attributes, 231–232

base-class specification, 171

class-base specification, 171

class-body, 171

class-member-declarations, 172–173

constants, 175–176

declarations, 169–170, 242–243

exception classes, 228

finalizers, 191

inheritance, 173–174

instance constructors, 190–191

instance members, 175 instance variables in, 86 interface types, 171

namespaces, organizing classes with, 161–162 naming, 341–342

new modifier, 174

overview, 27–28, 169

Current property

Index

partial declarations, 172

static constructors, 191

static members, 174

structs compared, 196–197

syntactic grammar, 296–307 **classifications of expressions, 111–112** class-member-declarations**, 172–173 closed constructed types, 243**

**closed type, 249**

**CLR (Common Language Runtime), 2, 359–360**

**code blocks, 144 comments**

delimited comments, 40–42

lexical grammar, 276–277

nesting comments, 43

overview, 40

single-line comments, 42–43

syntax in XML documentation comments, 363–365 tags in XML documentation comments, 365

**Common Type System (CTS), 359 compilation**

attributes, 237

conditional compilation directives, 54 conditional compilation symbols, 53 unsafe code, 273

**compilation units, 35–37, 162–163**

**compile-time errors, 259**

**conditional AND operator, 25, 114**

Conditional **attribute, 238–239 conditional attribute class, 239 conditional compilation directives, 54 conditional compilation symbols, 53 conditional logical operators, 133–134 conditional methods, 238**

**conditional operator, 25**

**conditional OR operator, 25, 114**

**const (keyword), 279**

**constant expressions, 135–138**

**constants, 28, 175–176**

**constraints, 253–255 constructed types**

alias directives, 250

closed type, 249

generics, 249–250

members, 250

open type, 249

overview, 249

type arguments, 249

**constructors, 199**

**continue (keyword), 279**

continue **statement**

overview, 157

rules for, 93

**conversion operators, 190 conversions**

anonymous method conversions, 109 explicit conversions, 22, 103–107

implicit conversions, 22, 99–103 method group conversions, 109 null type conversions, 109 nullable conversions, 109–110

overview, 99

standard conversions, 107–108

user-defined conversions, 108

**cost of using C#**

bare minimum to start with C# programming, 7–10 free tools, 10–13

high-end tools, 15–16

.NET Framework, 9–10

overview, 7

text editor, 8–9

UltraEdit, 13–15

Windows Notepad, 8–9

**Crimson Editor, 15**

**CTS (Common Type System), 359**

Current **property, 262**

decimal (keyword)

#### D

**decimal (keyword), 279 decimal type**

to float/double type conversion, 105 to integral type conversion, 104 overview, 20, 77

**declaration directives, 54 declaration statements**

local constant declarations, 147 local variable declarations, 146–147 overview, 146

rules for, 91

**declarations**

class declarations, 169–170, 242–243

delegate declarations, 222, 223, 248

enums, 216–217

interface declarations, 210–211, 247–248 local constant declarations, 147

local variable declarations, 146–147 namespace declarations, 58–60, 163–164

overview, 58–60

struct declarations, 247

type declarations, 166–167

**declared accessibility, 62–63**

**default constructors, 72**

**default (keyword), 279 default value expression, 130 default values, 89, 198**

**defining interfaces, 210 definite assignment**

block statements, rules for, 91 break statements, rules for, 93 checked statements, rules for, 91 continue statements, rules for, 93

declaration statements, rules for, 91

do statements, rules for, 92–93

double ampersand (&&) expressions, rules for, 96–97 double pipe (||) expressions, rules for, 97 exclamation point (!) expressions, rules for, 97 expression statements, rules for, 91

foreach statements, rules for, 94

goto statements, rules for, 93 if statements, rules for, 91–92 initially assigned variables, 90 initially unassigned variables, 90

lock statements, rules for, 94–95 overview, 89–90

question mark (?) expressions, rules for, 98

return statements, rules for, 93 rules for determining, 90–98

simple expressions, rules for, 95–96 statements, general rules for, 91 switch statements, rules for, 92 throw statements, rules for, 93

try-catch statements, rules for, 93–94 try-finally statements, rules for, 94 unchecked statements, rules for, 91 using statements, rules for, 94

while statements, rules for, 92

yield statements, rules for, 98

**delegate (keyword), 279**

delegate **type, 79 delegates**

declarations, 222, 223, 248

instantiation, 224–225

invocation list, 223–224

members, 62

modifiers, 222–223

overview, 31, 221–222

syntactic grammar, 308

**delimited comments, 40–42 destructors. *See* finalizers diagnostic directives, 54 directives**

conditional compilation directives, 54 conditional compilation symbols, 53 declaration directives, 54

diagnostic directives, 54

line directives, 55

overview, 51–53

pragma directives, 55

preprocessing expressions, 53

examples

Index

region control directives, 54–55 using, 36

Dispose **method, 262–263**

**do (keyword), 280**

do **statement**

overview, 154–155

rules for, 92–93

**DotGNU, 3**

**double ampersand (**&&**) expression, 96–97 double (keyword), 280**

**double pipe (**||**) expression, 97 double type**

to decimal type conversion, 105 to float type conversion, 105

to int type conversion, 105 overview, 20

#### E

**ECMA-334 C# Language Specification, 1, 3 ECMA (European Computer Manufacturers**

**Association), 3**

**EditPad Lite, 15**

**EditPad Pro, 15 element access**

array access, 128

base-access, 128–129

checked operator, 130

indexer access, 128

new operator, 129

overview, 127

sizeof operator, 129–130

this-access, 128

typeof operator, 129

unchecked operator, 130

**else (keyword), 280**

**empty statements, 145**

**end point, 142**

**enterprise tools**

overview, 15

Visual C#, 16

Visual Studio, 15–16

**entry point, 18, 57**

**enum (keyword), 280**

**enumerable interfaces, 259**

**enumerable objects, 263**

**enumeration members, 61**

**enumeration types, 77**

**enumerator interfaces, 259 enums**

circular references, avoiding, 219 declarations, 216–217

members, 218–219

modifiers, 217–218

naming, 342

operators, 220

overview, 31, 215–216

syntactic grammar, 308–309

System.Enum, 219

values, 219–220

**equality operators, 25, 114, 119**

**European Computer Manufacturers Association (ECMA), 3**

**event access, 122–123**

**event (keyword), 280 events**

abstract accessor, 185

field-like events, 185

instance events, 185

interfaces, 213

naming, 343

override accessor, 186

overview, 29, 184–185

sealed accessor, 186

static events, 185

virtual accessor, 185

**examples**

sample C# code, 3–5

of unsafe code, 269–270

exception classes

**exception classes, 228 exceptions**

catch clause, not finding, 229 handling, 229

overview, 227

System.ArithmeticException, 228

System.ArrayTypeMismatchException, 228

System.DivideByZeroException, 228

System.Exception, 228

System.IndexOutOfRangeException, 228

System.InvalidCastException, 228

System.NullReferenceException, 228

System.OutOfMemoryException, 228

System.OverflowException, 228

System.StackOverflowException, 228

System.TypeInitializationException, 228

throwing, 227

**exclamation point (**!**) expression, 97 execution interruptions for** MoveNext **method,**

**261–262**

**explicit base interfaces, 211 explicit conversions**

explicit enumeration conversions, 105 explicit numeric conversions, 103–105 explicit reference conversions, 106 explicit type parameter conversions, 107 overview, 103

standard explicit conversions, 108 unboxing conversions, 107

user-defined explicit conversions, 107

**explicit enumeration conversions, 105**

**explicit interface member implementations, 248 explicit (keyword), 280**

**explicit numeric conversions**

decimal type to float/double type conversion, 105 decimal type to integral type conversion, 104 double type to float type conversion, 105 float/double type to decimal type conversion, 105 float/double type to int type conversion, 105 integral type to integral type conversion, 104 overview, 103–104

**explicit reference conversions, 106 explicit type parameter conversions, 107**

**expression statements**

overview, 148

rules for, 91

**expressions**

anonymous methods, 131

arithmetic operators, 131–132

assignment operators, 135

Boolean expressions, 138

cast expressions, 131

classifications of, 111–112 conditional logical operators, 133–134 constant expressions, 135–138 default value expression, 130

function members, 121–125

logical operators, 133–134

member lookup, 119–120 null coalescing operator, 134 and operators, 112–119

overview, 24–26, 111

primary expressions, 125–130 relational/type testing operators, 132–133 results of, 112

shift operators, 132

syntactic grammar, 309–317

unary expressions, 131

values for, 112

**extern (keyword), 280**

extern-alias-directive**, 164–165**

#### F

**false (keyword), 280**

**field initialization, 178, 199**

**field-like events, 185 fields**

field initialization, 178

instance fields, 177

overview, 24, 28, 176–177

readonly fields, 177

static fields, 177

variable initialization, 178

volatile fields, 177–178

identity conversions

Index

**finalizers**

overview, 29, 191

structs, 199

**finally (keyword), 280**

**fixed (keyword), 280**

fixed **modifier, 270–272**

**float (keyword), 280 float type**

to decimal type conversion, 105 to int type conversion, 105 overview, 20

**floating-point types, 76–77**

**for (keyword), 280**

for **statement, 155–156**

**foreach (keyword), 280**

foreach **statement**

overview, 156

rules for, 94

**format standardization, 3**

**free tools, 10–13 function members**

argument lists, 124

event access, 122–123

indexer access, 123

instance constructor invocation, 123 method invocation, 121

operator invocation, 123

overload resolution, 125

overview, 121

property access, 122

#### G

**generic methods**

calling, 251–252

inference of type arguments, 251–252 overview, 250–251

signatures, 251

virtual generic methods, 251

**generics**

advantages of, 242

class declarations, 242–243

class members, 245

constraints, 253–255

constructed types, 249–250

delegate declarations, 248

explicit interface member implementations, 248 instance type, 244–245

interface declarations, 247–248 operators in generic classes, 247 overloading in generic classes, 246–247 overview, 31–32, 241

protected members, access to, 246 static constructors, 246

static fields, 246

struct declarations, 247

syntactic grammar, 317–318 templates in C++ compared, 241 type parameters, 243–244 where not to use, 252–253

GetEnumerator **method, 263–264**

**global attributes, 36**

**goto (keyword), 280**

goto **statement**

overview, 157

rules for, 93

**grammar**

ambiguities, 37–38

lexical grammar, 37, 39–40, 275–292

syntactic grammar, 37, 292–331 unsafe code, extensions for, 332–335

#### H

**Hajlsberg, Anders (C# principal designer), 2 “Hello, World!” program, 3–5, 17–18**

**high-end tools, 15–16 history of C#, 2**

#### I

**IDE (Integrated Development Environment), 35 identifiers, 44–45, 277–279**

**identity conversions, 100**

if (keyword)

**if (keyword), 280**

if **statement**

overview, 148–149

rules for, 91–92

**implementation-defined behavior, 360–361 implementations in development for C#, 3 implicit constant expression conversions, 103 implicit conversions**

boxing conversions, 102

identity conversions, 100

implicit constant expression conversions, 103 implicit enumeration conversions, 101 implicit numeric conversions, 100

implicit reference conversions, 101–102 implicit type parameter conversions, 102–103 overview, 99–100

standard implicit conversions, 107–108 user defined implicit conversions, 103 user-defined implicit conversions, 108

**implicit enumeration conversions, 101 implicit (keyword), 280**

**implicit numeric conversions, 100 implicit reference conversions, 101–102**

**implicit type parameter conversions, 102–103 in (keyword), 280**

**index signatures, 63**

**indexer access, 123, 128 indexers**

overloading, 64

overview, 29, 186–187

**inference of type arguments, 251–252 inheritance**

overview, 30, 173–174

structs, 197

**initializers, 206–208**

**initially assigned variables, 90 initially unassigned variables, 90 instance constructors**

invocation, 123

overloading, 64

overview, 29, 190–191

signatures, 63

**instance events, 185**

**instance fields, 177**

**instance members, 175**

**instance methods, 180**

**instance properties, 182**

**instance type, 244–245 instance variables**

in classes, 86

overview, 86

in structs, 87

**instantiation, 224–225**

**int (keyword), 280**

int **type, 20, 74–75**

**integer literals, 47 integral types**

byte type, 74–75

char type, 74–75

int type, 74–75

to integral type conversion, 104

long type, 74–75

overview, 74–76

sbyte type, 74–75

short type, 74–75

uint type, 74–75

ulong type, 74–75

ushort type, 74–75

**Integrated Development Environment (IDE), 35 interface (keyword), 280**

**interface types, 171 interfaces**

body, 212

declarations, 210–211, 247–248

defining, 210

events, 213

explicit base interfaces, 211 members, 62, 212

methods, 212

modifiers, 211

naming, 342

overview, 30–31, 209

properties, 212–213

structs, 195

syntactic grammar, 318–320

**internal (keyword), 280**

Index

**interoperability with legacy code, 359–360 invocation expressions, 127**

**invocation list, 223–224**

**is (keyword), 280**

**ISO/IEC 23270 standard, 3 iteration statements**

do statement, 154–155

for statement, 155–156

foreach statement, 156

overview, 154

while statement, 154

**iterator block, 258–259 iterators**

compile-time errors, 259

Current property, 262

Dispose method, 262–263

enumerable interfaces, 259

enumerable objects, 263

enumerator interfaces, 259

execution interruptions for MoveNext method, 261–262

GetEnumerator method, 263–264

iterator block, 258–259

MoveNext method, 260–262

overview, 32, 257–258

this, 260

yield break statement, 262 yield return statement, 261 yield type, 260

#### J

**jagged arrays, 22–23 jump statements**

break statement, 156–157

continue statement, 157

goto statement, 157

overview, 156

return statement, 158

throw statement, 158

#### K

**keywords**

abstract, 279

as, 279

to avoid, 339–341

base, 279

bool, 279

break, 279

byte, 279

case, 279

catch, 279

char, 279

checked, 279

class, 279

const, 279

continue, 279

decimal, 279

default, 279

delegate, 279

do, 280

double, 280

else, 280

enum, 280

event, 280

explicit, 280

extern, 280

false, 280

finally, 280

fixed, 280

float, 280

for, 280

foreach, 280

goto, 280

if, 280

implicit, 280

in, 280

int, 280

interface, 280

internal, 280

is, 280

lexical grammar, 279–282

lock, 280

keywords

keywords

**keywords (continued)**

long, 280

namespace, 280

new, 280

null, 281

object, 281

operator, 281

out, 281

override, 281

overview, 46

params, 281

private, 281

protected, 281

public, 281

readonly, 281

ref, 281

return, 281

sbyte, 281

sealed, 281

short, 281

sizeof, 281

stackalloc, 281

static, 281

string, 281

struct, 281

switch, 281

this, 281

throw, 281

true, 281

try, 281

typeof, 282

uint, 282

ulong, 282

unchecked, 282

unsafe, 282

ushort, 282

using, 282

virtual, 282

void, 282

volatile, 282

while, 282

**L**

**labeled statements, 145–146 language structure**

attributes, 36

comments, 40–43

compilation units, 35–37

directives, 36, 51–55

global attributes, 36

grammar, 37–40

line terminators, 40

namespace member declarations, 36 source files, 35–37

tokens, 36, 43–51

whitespace, 43

**learning C#, 5 lexical grammar**

comments, 276–277

identifiers, 277–279

line terminators, 282

literals, 282–286

operators/punctuators, 286–288

overview, 36, 39–40, 275–276

pre-processing directives, 288–292 unicode escape characters, 292 white space, 292

**lexical grammar (keyword), 279–282 lifted operators**

equality operators, 119

overview, 118

relational operators, 119

unary operators, 118

**line directives, 55**

**line terminators, 40, 282 literals**

Boolean literals, 47

character literals, 48–49

integer literals, 47

lexical grammar, 282–286

null literal, 49

overview, 46

namespaces

Index

real literals, 48

string literals, 49

**local constant declarations, 147 local variable declarations, 146–147 local variables, 88–89**

**lock (keyword), 280**

lock **statements, rules for, 94–95 logical AND operator, 25, 114**

**logical operators, 133–134**

**logical OR operator, 25, 114**

**logical XOR operator, 25, 114**

**long (keyword), 280**

long **type, 21, 74–75**

#### M

**member access, 62, 126–127 member lookup**

base types, 120

overview, 119–120

**members**

array members, 62, 205

class members, 61–62

constructed types, 250

delegate members, 62

enumeration members, 61, 218–219

interface members, 62, 212

namespace members, 61

overview, 60–61, 172–173, 245

struct members, 61, 195–196

**memory management, 66–67 method group conversions, 109 methods**

abstract methods, 181

body, 181

instance methods, 180

interfaces, 212

invocation, 121

method body, 181

naming, 343

overloading, 64

override methods, 180–181

overview, 28, 178–179

parameters, 179–180

sealed methods, 181

signatures, 63–64

static methods, 180

virtual methods, 180

**Microsoft Intermediate Language (MSIL), 36, 359–360 modifiers**

delegates, 222–223

enums, 217–218

interfaces, 211

overview, 170–171

structs, 195

**Mono, 3**

MoveNext **method, 260–262**

**multiplicative operators, 25, 114**

#### N

**the name C#, explanation of, 1 named parameters, 232**

**namespace (keyword), 280**

namespace-member-declaration**, 36, 166 namespaces**

compilation units, 162–163

declarations, 58–60, 163–164

extern-alias-directive, 164–165

members, 61

namespace-member-declaration, 166

naming, 341

organizing classes with, 161–162 overview, 66, 161

qualified-alias-member, 167–168 scope controlled with, 162

type declarations, 166–167

using directives, 165–166

naming conventions

**naming conventions**

abbreviations, 339

attribute naming, 342

Camel case, 338

capitalization, 337

case sensitivity, 338–339

class naming, 341–342 enumeration type naming, 342 event naming, 343

interface naming, 342 keywords to avoid, 339–341 method naming, 343

namespace naming, 341

overview, 337

parameter naming, 343

Pascal case, 337

property naming, 343 static field naming, 342 uppercase, 338

**nesting comments, 43**

**.NET, 2–3**

**.NET Framework, 9–10**

**new (keyword), 280**

new **modifier, 174**

new **operator, 129**

**Notepad ++, 15**

**NoteTab, 15**

**null coalescing operators, 114, 134**

**null (keyword), 281**

**null literal, 49**

null **type, 79**

**null type conversions, 109 nullable conversions, 109–110**

**nullable types, 32, 80–81**

#### O

**object (keyword), 281**

object **type, 21, 79**

**objects, how types are treated as, 70**

Obsolete **attribute, 239–240**

**open constructed type, 243, 249**

**operator (keyword), 281**

**operators. *See also* unary operators**

additive operators, 25, 114

assignment operators, 115

binary operator, 113, 189

binary operator overload resolution, 118 conditional AND operator, 25, 114

conditional operator, 25

conditional OR operator, 25, 114

conversion operators, 190

enums, 220

equality operators, 25, 114, 119

and expressions, 112–119 in generic classes, 247 invocation, 123

lifted operators, 118–119

list of, 286–288

logical AND operator, 25, 114

logical OR operator, 25, 114

logical XOR operator, 25, 114

multiplicative operators, 25, 114 null coalescing operators, 114 overloading, 64, 115–118

overview, 29, 112–113, 187–189

pointers, 268–269

precedence, 113–115

primary operators, 25, 114

relational operators, 119

relational/type-testing operators, 25, 114

shift operators, 25, 114

signatures, 64

ternary operator, 113

tokens, 50–51

types of, 113

**out (keyword), 281**

out **parameter, 70**

**output parameters, 24, 88, 180**

**overload resolution, 125 overloading**

in generic classes, 246–247 indexers, 64

instance constructors, 64

methods, 64

operators, 64, 115–118

overview, 64

override **accessor**

events, 186

properties, 183–184

**override (keyword), 281**

**override methods, 180–181 overview of C#**

accessibility, 28

advantages of C#, 1–2, 5

classes, 27–28

#### P

**parameters**

arrays, 24, 180

attribute parameters, 233

for methods, 179–180

naming, 343

output parameters, 24

overview, 24

reference parameters, 24

value parameters, 24

**params (keyword), 281**

private (keyword)

constants, 28

Index

delegates, 31

enums, 31

events, 29

expressions, 24–26

fields, 24, 28

finalizers, 29

generics, 31–32

“Hello, World!” program, 17–18 history of C#, 2

implementations in development for C#, 3 indexers, 29

inheritance, 30

instance constructors, 29

interfaces, 30–31

iterators, 32

learning C#, 5

methods, 28

nullable types, 32

operators, 29

parameters, 24

properties, 28–29

source code, examining, 18 standardization, 3

statements, 26–27

static classes, 30

static constructors, 30

structs, 30

types, 19–23

variables, 23–24

**parenthesized expressions, 126**

partial **declarations, 172**

**Pascal case, 337**

**platform/OS portability, 359**

**pointers, 268–269 portability**

implementation-defined behavior, 360–361 interoperability with legacy code, 359–360 overview, 359, 362

platform/OS portability, 359

simplified deployment, 359

undefined behavior, 360

unspecified behavior, 361–362

**positional parameters, 232**

**pragma directives, 55**

**precedence, 24–26, 113–115**

**predefined reference types, 19, 21**

**predefined types, 19–21 predefined value types, 20–21**

**pre-processing directives, 288–292**

**preprocessing expressions, 53 primary expressions**

element access, 127–130

invocation expressions, 127

literals, 125

member access, 126–127

overview, 125

parenthesized expressions, 126

simple names, 126

**primary operators, 25, 114**

**private (keyword), 281**

Programmer’s Notepad

**Programmer’s Notepad, 15 properties**

abstract accessor, 183

access, 122

accessors, 182–184

instance properties, 182

interfaces, 212–213

naming, 343

override accessor, 183–184

overview, 28–29, 181–182

sealed accessor, 184

static properties, 182

virtual accessor, 183

**protected (keyword), 281 protected members, access to, 246 public (keyword), 281**

**punctuators**

list of, 286–288

overview, 50–51

#### Q

qualified-alias-member**, 167–168 question mark (**?**) expressions, rules for, 98**

#### R

**reachability, 142–144**

readonly **fields, 177**

**readonly (keyword), 281**

**real literals, 48**

**rectangular arrays, 22–23**

**ref (keyword), 281**

ref **parameter, 70**

**reference parameters, 24, 87–88, 180 reference types**

array type, 79

boxing, 80

class type, 79

delegate type, 79

null type, 79

object type, 79

overview, 78

string type, 79

unboxing, 80

value types compared, 69–70 **region control directives, 54–55 relational operators, 119**

**relational/type-testing operators, 25, 114, 132–133**

**reserved attributes, 238**

**return (keyword), 281**

return **statement**

overview, 158

rules for, 93

**runtime retrieval of attribute instances, 237**

#### S

**sbyte (keyword), 281**

sbyte **type, 21, 74–75 scope**

controlled with namespaces, 162 overview, 64–66

sealed **accessor**

events, 186

properties, 184

**sealed (keyword), 281**

**sealed methods, 181 selection statements**

if statement, 148–149

overview, 148

switch statement, 149–153

**shift operators, 25, 114, 132**

**short (keyword), 281**

short **type, 21, 74–75 signatures**

generic methods, 251

index signatures, 63

instance constructor signatures, 63 method signatures, 63–64

operator signatures, 64

and overloading, 64

overview, 63

**simple expressions, rules for, 95–96**

switch statement

Index

**simple names, 126**

**simple types, 73**

**simplified deployment, 359**

**single-line comments, 42–43**

**sizeof (keyword), 281**

sizeof **operator, 129–130, 272 source code, examining, 18 source files, 35–37**

**specification, 233–236**

stackalloc**, 273**

**stackalloc (keyword), 281**

**standard conversions, 107–108 standard explicit conversions, 108 standard implicit conversions, 107–108 standard library, 345–357**

**standardization, 3**

**statement lists, 144–145 statements**

code blocks, 144

declaration statements, 146–147

empty statements, 145

end point, 142

expression statements, 148 general rules for, 91

iteration statements, 154–156

jump statements, 156–158

labeled statements, 145–146

overview, 26–27, 139–141

reachability, 142–144

selection statements, 148–153

statement lists, 144–145

syntactic grammar, 320–327

types of, 141

using statement, 158–159

yield statement, 159–160

**static classes, 30 static constructors**

generics, 246

overview, 30, 191

structs, 199

**static events, 185**

**static fields**

naming, 342

overview, 177, 246

**static (keyword), 281**

**static members, 174**

**static methods, 180**

**static properties, 182**

**static variables, 85**

**string (keyword), 281**

**string literals, 49**

string **type, 21, 79**

**struct declarations, 247**

**struct (keyword), 281**

**struct members, 61**

**struct types, 72–73**

struct-declaration**, 194–195 structs**

assignments, 198

body, 195

boxing, 198

class compared, 196–197

constructors, 199

default values, 198

field initializers, 199

finalizers, 199

inheritance, 197 instance variables in, 87 interfaces, 195

members, 195–196

modifiers, 195

overview, 30, 193–194

static constructors, 199

struct-declaration, 194–195

syntactic grammar, 327–329

this variable, 198

unboxing, 198

when to use, 199–200 **switch (keyword), 281** switch **statement**

overview, 149–153

rules for, 92

syntactic grammar

**syntactic grammar**

arrays, 293–294

attributes, 294–296

classes, 296–307

delegates, 308

enums, 308–309

expressions, 309–317

generics, 317–318

interfaces, 318–320

overview, 37, 292–293

statements, 320–327

structs, 327–329

types, 329–331

variables, 331

**syntax in XML documentation comments, 363–365**

System.ArithmeticException**, 228**

System.Array **type, 204**

System.ArrayTypeMismatchException**, 228**

System.DivideByZeroException**, 228**

System.Enum**, 219**

System.Exception**, 228**

System.IndexOutOfRangeException**, 228**

System.InvalidCastException**, 228**

System.NullReferenceException**, 228**

System.OutOfMemoryException**, 228**

System.OverflowException**, 228**

System.StackOverflowException**, 228**

System.TypeInitializationException**, 228**

System.ValueType **class, 71–72**

#### T

**tags in XML documentation comments, 365 templates in C++ compared to generics, 241 ternary operator, 113**

**text editor, 8–9**

**this (keyword), 281**

this **variable, 198, 260**

this-access**, 128**

**throw (keyword), 281**

throw **statement**

overview, 158

rules for, 93

**throwing exceptions, 227 tokens**

identifiers, 44–45

keywords, 46

literals, 46–49

operators, 50–51

overview, 43

punctuators, 50–51

Unicode escape sequences, 43–44

**trailing commas, 208**

**transformation, 36**

**true (keyword), 281**

**try (keyword), 281**

try-catch **statements, rules for, 93–94** try-finally **statements, rules for, 94 typeof (keyword), 282**

typeof **operator, 129 types**

arguments, 249

array types, 22–23, 79

bool type, 20, 77

boxing, 80

byte type, 20, 74–75

char type, 20, 74–75

class type, 79

conversions, 22

decimal type, 20

declarations, 166–167

double type, 20

float type, 20

instance type, 244–245

int type, 20, 74–75

long type, 21, 74–75

names, 66

object type, 21

objects, how types are treated as, 70

out parameter, 70

overloading, 22

overview, 19, 19–23, 74–76

parameters, 243–244

predefined reference types, 19, 21

predefined types, 19–21 predefined value types, 20–21

value types

Index

ref parameter, 70

reference types, 19, 69–70, 78–80

sbyte type, 21, 74–75

short type, 21, 74–75

string type, 21

syntactic grammar, 329–331

types of, 69

uint type, 21, 74–75

ulong type, 21, 74–75

unboxing, 80

ushort type, 21, 74–75

using, 76

value types, 19, 69–77, 80–81

#### U

**uint (keyword), 282**

uint **type, 21, 74–75**

**ulong (keyword), 282**

ulong **type, 21, 74–75**

**UltraEdit, 13–15**

**unary expressions, 131 unary operators**

lifted operators, 118

overload resolution, 117

overview, 25, 113, 189

precedence, 114

**unboxing conversions, 80, 107, 198**

**unchecked (keyword), 282**

unchecked **operator, 130** unchecked **statements, rules for, 91 undefined behavior, 360**

**unicode escape characters, 43–44, 292 unsafe code**

advantages of, 266

compilation, 273

disadvantages of, 266

example, 269–270

extensions for, 332–335

fixed modifier, 270–272

overview, 265

pointers, 268–269

sizeof operator, 272

stackalloc, 273

unsafe contexts, 266–268

**unsafe contexts, 266–268**

**unsafe (keyword), 282**

**unspecified behavior, 361–362**

**unwrapping, 109**

**uppercase, 338**

**user-defined conversions, 103, 107, 108 user-defined explicit conversions, 107**

**user-defined implicit conversions, 103, 108**

**ushort (keyword), 282**

ushort **type, 21, 74–75**

using **directives**

overview, 165

using-alias-directive, 165

using-namespace-directive, 166

**using (keyword), 282**

using **statement**

overview, 158–159

rules for, 94

using-alias-directive**, 165**

using-namespace-directive**, 166**

#### V

**value parameters, 24, 87, 180 value types**

bool type, 77

boxing, 80

decimal type, 77

default constructors, 72

enumeration types, 77

floating-point types, 76–77

integral types, 74–76

nullable types, 80–81

overview, 70–71

reference types compared, 69–70 simple types, 73

struct types, 72–73

System.ValueType class, 71–72

unboxing, 80

variable initialization

**variable initialization, 178 variables**

array elements, 85–86

categories of, 84–85

default values, 89

definite assignment, 89–98

instance variables, 86–87

local variables, 88–89

output parameters, 88

overview, 23–24, 83–84

reference parameters, 87–88

static variables, 85

syntactic grammar, 331

value parameter, 87

virtual **accessor**

events, 185

properties, 183

**virtual generic methods, 251 virtual (keyword), 282**

**virtual methods, 180**

**Visual Studio, 15–16**

**void (keyword), 282**

**void pointers, 268**

**volatile (keyword), 282**

**volatile fields, 177–178**

**W**

**while (keyword), 282**

while **statement**

overview, 154

rules for, 92

**whitespace, 43, 292 Windows Notepad**

overview, 8–9

writing code in, 10–13

**wrapping, 109 writing code**

with free tools, 10–13

in Windows Notepad, 10–13

#### X

**XML documentation comments**

overview, 363

syntax, 363–365

tags, 365

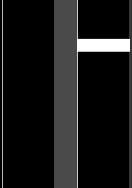
#### Y

yield break **statement, 262** yield return **statement, 261** yield **statement**

overview, 159–160

rules for, 98

**yield type, 260**



you can access more than 70 complete Wrox books online, wherever you happen to be! Every diagram, description, screen capture, and code sample is available with your subscription to the Wrox Reference Library. For answers when and where you need them, go to wrox.books24x7.com and subscribe today!

* Mac • Web
* Microsoft Office • XML
* .NET
* Open Source
* PHP/MyseL
* SQL Server
* Visual Basic

AsP.NET

C’/ C

* General
* Java

Find books on

wherever

Get

# at

**Programmer to ProgrammerTM**

**P2P FORUM**

**FREE NEWSLETTER**

**ABOUT WROX**

more Wrox

# Wrox.com!

**BROWSE BOOKS**

###### Special Deals

Take advantage of special offers every month

###### Unlimited Access. . .

. . . to over 70 of our books in the Wrox Reference Library. (see more details on-line)

###### Free Chapter Excerpts

Be the first to preview chapters from the latest Wrox publications

###### Forums, Forums, Forums

Take an active role in online discussions with fellow programmers

###### Meet Wrox Authors!

Read running commentaries from authors on their programming experiences and whatever else they want to talk about

###### Browse Books

.NET

SQL Server Java

XML

Visual Basic C#/ C++



**Join the community!**

Sign-up for our free monthly newsletter at

**newsletter.wrox.com**