



C#

IN DEPTH

THIRD EDITION

Jon Skeet

FOREWORD BY ERIC LIPPERT

 MANNING

Praise for the Second Edition

A masterpiece about C#.

—Kirill Osenkov, Microsoft C# Team

If you are looking to master C# then this book is a must-read.

—Tyson S. Maxwell
Sr. Software Engineer, Raytheon

We're betting that this will be the best C# 4.0 book out there.

—Nikander Bruggeman and Margriet Bruggeman
.NET consultants, Lois & Clark IT Services

A useful and engaging insight into the evolution of C# 4.

—Joe Albahari
Author of *LINQPad* and *C# 4.0 in a Nutshell*

One of the best C# books I have ever read.

—Aleksey Nudelman
CEO, C# Computing, LLC

This book should be required reading for all professional C# developers.

—Stuart Caborn
Senior Developer, BNP Paribas

A highly focused, master-level resource on language updates across all major C# releases. This book is a must-have for the expert developer wanting to stay current with new features of the C# language.

—Sean Reilly, Programmer/Analyst
Point2 Technologies

Why read the basics over and over again? Jon focuses on the chewy, new stuff!

—Keith Hill, Software Architect
Agilent Technologies

Everything you didn't realize you needed to know about C#.

—Jared Parsons
Senior Software Development Engineer
Microsoft

Praise for the First Edition

Simply put, C# in Depth is perhaps the best computer book I've read.

—Craig Pelkie, Author, *System iNetwork*

I have been developing in C# from the very beginning and this book had some nice surprises even for me. I was especially impressed with the excellent coverage of delegates, anonymous methods, covariance and contravariance. Even if you are a seasoned developer, C# in Depth will teach you something new about the C# language... This book truly has depth that no other C# language book can touch.

—Adam J. Wolf
Southeast Valley .NET User Group

I enjoyed reading the whole book; it is well-written—the samples are easy to understand. I actually found it very easy to engage into the whole lambda expressions topic and really liked the chapter about lambda expressions.

—Jose Rolando Guay Paz
Web Developer, CSW Solutions

This book wraps up the author's great knowledge of the inner workings of C# and hands it over to readers in a well-written, concise, usable book.

—Jim Holmes
Author of *Windows Developer Power Tools*

Every term is used appropriately and in the right context, every example is spot-on and contains the least amount of code that shows the full extent of the feature...this is a rare treat.

—Franck Jeannin, Amazon UK reviewer

If you have developed using C# for several years now, and would like to know the internals, this book is absolutely right for you.

—Golo Roden
Author, Speaker, and Trainer for .NET
and related technologies

The best C# book I've ever read.

—Chris Mullins, C# MVP

C# in Depth

THIRD EDITION

JON SKEET



MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity.

For more information, please contact


Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964.

©2014 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

- ⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

 Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964

Development editor: Jeff Bleiel
Copyeditor: Andy Carroll
Proofreader: Katie Tennant
Typesetter: Dottie Marsico
Cover designer: Marija Tudor

ISBN 9781617291340

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – MAL – 18 17 16 15 14 13

To my boys, Tom, Robin, and William

brief contents

PART 1	PREPARING FOR THE JOURNEY.....	1
1	■ The changing face of C# development	3
2	■ Core foundations: building on C# 1	29
PART 2	C# 2: SOLVING THE ISSUES OF C# 1	57
3	■ Parameterized typing with generics	59
4	■ Saying nothing with nullable types	105
5	■ Fast-tracked delegates	133
6	■ Implementing iterators the easy way	159
7	■ Concluding C# 2: the final features	182
PART 3	C# 3: REVOLUTIONIZING DATA ACCESS	205
8	■ Cutting fluff with a smart compiler	207
9	■ Lambda expressions and expression trees	232
10	■ Extension methods	262
11	■ Query expressions and LINQ to Objects	285
12	■ LINQ beyond collections	328

PART 4	C# 4: PLAYING NICELY WITH OTHERS	369
13	■ Minor changes to simplify code	371
14	■ Dynamic binding in a static language	409
PART 5	C# 5: ASYNCHRONY MADE SIMPLE	461
15	■ Asynchrony with async/await	463
16	■ C# 5 bonus features and closing thoughts	519

contents

foreword xix
preface xxi
acknowledgments xxii
about this book xxiv
about the author xxix
about the cover illustration xxx

PART 1 PREPARING FOR THE JOURNEY1

1 The changing face of C# development 3

1.1 Starting with a simple data type 4

The Product type in C# 1 5 ■ *Strongly typed collections in C#* 2 6
Automatically implemented properties in C# 3 7 ■ *Named arguments in C#* 4 8

1.2 Sorting and filtering 9

Sorting products by name 9 ■ *Querying collections* 12

1.3 Handling an absence of data 14

Representing an unknown price 14 ■ *Optional parameters and default values* 16

1.4 Introducing LINQ 16

Query expressions and in-process queries 17 ■ *Querying XML* 18 ■ *LINQ to SQL* 19

- 1.5 COM and dynamic typing 20
 - Simplifying COM interoperability* 20
 - *Interoperating with a dynamic language* 21
- 1.6 Writing asynchronous code without the heartache 22
- 1.7 Dissecting the .NET platform 23
 - C#, the language* 24
 - *Runtime* 24
 - *Framework libraries* 24
- 1.8 Making your code super awesome 25
 - Presenting full programs as snippets* 25
 - *Didactic code isn't production code* 26
 - *Your new best friend: the language specification* 27
- 1.9 Summary 28

2 Core foundations: building on C# 1 29

- 2.1 Delegates 30
 - A recipe for simple delegates* 30
 - *Combining and removing delegates* 35
 - *A brief diversion into events* 36
 - *Summary of delegates* 37
- 2.2 Type system characteristics 38
 - C#'s place in the world of type systems* 38
 - *When is C# 1's type system not rich enough?* 41
 - *Summary of type system characteristics* 44
- 2.3 Value types and reference types 44
 - Values and references in the real world* 45
 - *Value and reference type fundamentals* 46
 - *Dispelling myths* 47
 - *Boxing and unboxing* 49
 - *Summary of value types and reference types* 50
- 2.4 Beyond C# 1: new features on a solid base 51
 - Features related to delegates* 51
 - *Features related to the type system* 53
 - *Features related to value types* 55
- 2.5 Summary 56

PART 2 C# 2: SOLVING THE ISSUES OF C# 157

3 Parameterized typing with generics 59

- 3.1 Why generics are necessary 60
- 3.2 Simple generics for everyday use 62
 - Learning by example: a generic dictionary* 62
 - *Generic types and type parameters* 64
 - *Generic methods and reading generic declarations* 67

- 3.3 Beyond the basics 70
 - Type constraints* 71 ■ *Type inference for type arguments of generic methods* 76 ■ *Implementing generics* 77
- 3.4 Advanced generics 83
 - Static fields and static constructors* 84 ■ *How the JIT compiler handles generics* 85 ■ *Generic iteration* 87 ■ *Reflection and generics* 90
- 3.5 Limitations of generics in C# and other languages 94
 - Lack of generic variance* 94 ■ *Lack of operator constraints or a “numeric” constraint* 99 ■ *Lack of generic properties, indexers, and other member types* 101 ■ *Comparison with C++ templates* 101 ■ *Comparison with Java generics* 103
- 3.6 Summary 104

4 *Saying nothing with nullable types* 105

- 4.1 What do you do when you just don't have a value? 106
 - Why value type variables can't be null* 106
 - Patterns for representing null values in C# 1* 107
- 4.2 System.Nullable<T> and System.Nullable 109
 - Introducing Nullable<T>* 109 ■ *Boxing Nullable<T> and unboxing* 112 ■ *Equality of Nullable<T> instances* 113
 - Support from the nongeneric Nullable class* 114
- 4.3 C# 2's syntactic sugar for nullable types 114
 - The ? modifier* 115 ■ *Assigning and comparing with null* 116
 - Nullable conversions and operators* 118 ■ *Nullable logic* 121
 - Using the as operator with nullable types* 123 ■ *The null coalescing operator* 123
- 4.4 Novel uses of nullable types 126
 - Trying an operation without using output parameters* 127
 - Painless comparisons with the null coalescing operator* 129
- 4.5 Summary 131

5 *Fast-tracked delegates* 133

- 5.1 Saying goodbye to awkward delegate syntax 134
- 5.2 Method group conversions 136
- 5.3 Covariance and contravariance 137
 - Contravariance for delegate parameters* 138 ■ *Covariance of delegate return types* 139 ■ *A small risk of incompatibility* 141

- 5.4 Inline delegate actions with anonymous methods 142
 - Starting simply: acting on a parameter* 142
 - *Returning values from anonymous methods* 145
 - *Ignoring delegate parameters* 146
- 5.5 Capturing variables in anonymous methods 148
 - Defining closures and different types of variables* 148
 - Examining the behavior of captured variables* 149
 - *What's the point of captured variables?* 151
 - *The extended lifetime of captured variables* 152
 - *Local variable instantiations* 153
 - Mixtures of shared and distinct variables* 155
 - *Captured variable guidelines and summary* 156
- 5.6 Summary 158

6 *Implementing iterators the easy way* 159

- 6.1 C# 1: The pain of handwritten iterators 160
- 6.2 C# 2: Simple iterators with yield statements 163
 - Introducing iterator blocks and yield return* 163
 - *Visualizing an iterator's workflow* 165
 - *Advanced iterator execution flow* 167
 - *Quirks in the implementation* 170
- 6.3 Real-life iterator examples 172
 - Iterating over the dates in a timetable* 172
 - *Iterating over lines in a file* 173
 - *Filtering items lazily using an iterator block and a predicate* 176
- 6.4 Pseudo-synchronous code with the Concurrency and Coordination Runtime 178
- 6.5 Summary 180

7 *Concluding C# 2: the final features* 182

- 7.1 Partial types 183
 - Creating a type with multiple files* 184
 - *Uses of partial types* 186
 - *Partial methods—C# 3 only!* 188
- 7.2 Static classes 190
- 7.3 Separate getter/setter property access 192
- 7.4 Namespace aliases 193
 - Qualifying namespace aliases* 194
 - *The global namespace alias* 195
 - *Extern aliases* 196

- 7.5 Pragma directives 197
 - Warning pragmas* 197 ■ *Checksum pragmas* 198
- 7.6 Fixed-size buffers in unsafe code 199
- 7.7 Exposing internal members to selected assemblies 201
 - Friend assemblies in the simple case* 201 ■ *Why use InternalsVisibleTo?* 202 ■ *InternalsVisibleTo and signed assemblies* 203
- 7.8 Summary 204

PART 3 C# 3: REVOLUTIONIZING DATA ACCESS.....205

8 *Cutting fluff with a smart compiler* 207

- 8.1 Automatically implemented properties 208
- 8.2 Implicit typing of local variables 211
 - Using var to declare a local variable* 211 ■ *Restrictions on implicit typing* 213 ■ *Pros and cons of implicit typing* 214
 - Recommendations* 215
- 8.3 Simplified initialization 216
 - Defining some sample types* 216 ■ *Setting simple properties* 217
 - Setting properties on embedded objects* 219 ■ *Collection initializers* 220 ■ *Uses of initialization features* 223
- 8.4 Implicitly typed arrays 224
- 8.5 Anonymous types 225
 - First encounters of the anonymous kind* 225 ■ *Members of anonymous types* 227 ■ *Projection initializers* 228 ■ *What's the point?* 229
- 8.6 Summary 231

9 *Lambda expressions and expression trees* 232

- 9.1 Lambda expressions as delegates 234
 - Preliminaries: Introducing the Func<...> delegate types* 234
 - First transformation to a lambda expression* 235 ■ *Using a single expression as the body* 236 ■ *Implicitly typed parameter lists* 236
 - Shortcut for a single parameter* 237
- 9.2 Simple examples using List<T> and events 238
 - Filtering, sorting, and actions on lists* 238 ■ *Logging in an event handler* 240

- 9.3 Expression trees 241
 - Building expression trees programmatically* 242
 - *Compiling expression trees into delegates* 243
 - *Converting C# lambda expressions to expression trees* 244
 - *Expression trees at the heart of LINQ* 248
 - *Expression trees beyond LINQ* 249
- 9.4 Changes to type inference and overload resolution 251
 - Reasons for change: streamlining generic method calls* 252
 - Inferred return types of anonymous functions* 253
 - *Two-phase type inference* 254
 - *Picking the right overloaded method* 258
 - Wrapping up type inference and overload resolution* 260
- 9.5 Summary 260

10 *Extension methods* 262

- 10.1 Life before extension methods 263
- 10.2 Extension method syntax 265
 - Declaring extension methods* 265
 - *Calling extension methods* 267
 - *Extension method discovery* 268
 - *Calling a method on a null reference* 269
- 10.3 Extension methods in .NET 3.5 271
 - First steps with Enumerable* 271
 - *Filtering with Where and chaining method calls together* 273
 - *Interlude: haven't we seen the Where method before?* 275
 - *Projections using the Select method and anonymous types* 276
 - *Sorting using the OrderBy method* 277
 - *Business examples involving chaining* 278
- 10.4 Usage ideas and guidelines 280
 - "Extending the world" and making interfaces richer* 280
 - *Fluent interfaces* 280
 - *Using extension methods sensibly* 282
- 10.5 Summary 284

11 *Query expressions and LINQ to Objects* 285

- 11.1 Introducing LINQ 286
 - Fundamental concepts in LINQ* 286
 - *Defining the sample data model* 291
- 11.2 Simple beginnings: selecting elements 292
 - Starting with a source and ending with a selection* 293
 - *Compiler translations as the basis of query expressions* 293
 - *Range variables and nontrivial projections* 296
 - *Cast, OfType, and explicitly typed range variables* 298

- 11.3 Filtering and ordering a sequence 300
 - Filtering using a where clause* 300
 - *Degenerate query expressions* 301
 - *Ordering using an orderby clause* 302
- 11.4 Let clauses and transparent identifiers 304
 - Introducing an intermediate computation with let* 305
 - Transparent identifiers* 306
- 11.5 Joins 307
 - Inner joins using join clauses* 307
 - *Group joins with join...into clauses* 311
 - *Cross joins and flattening sequences using multiple from clauses* 314
- 11.6 Groupings and continuations 318
 - Grouping with the group...by clause* 318
 - *Query continuations* 321
- 11.7 Choosing between query expressions and dot notation 324
 - Operations that require dot notation* 324
 - *Query expressions where dot notation may be simpler* 325
 - *Where query expressions shine* 325
- 11.8 Summary 326

12 LINQ beyond collections 328

- 12.1 Querying a database with LINQ to SQL 329
 - Getting started: the database and model* 330
 - *Initial queries* 332
 - *Queries involving joins* 334
- 12.2 Translations using IQueryable and IQueryProvider 336
 - Introducing IQueryable<T> and related interfaces* 337
 - *Faking it: interface implementations to log calls* 338
 - *Gluing expressions together: the Queryable extension methods* 341
 - *The fake query provider in action* 342
 - *Wrapping up IQueryable* 344
- 12.3 LINQ-friendly APIs and LINQ to XML 344
 - Core types in LINQ to XML* 345
 - *Declarative construction* 347
 - Queries on single nodes* 349
 - *Flattened query operators* 351
 - Working in harmony with LINQ* 352
- 12.4 Replacing LINQ to Objects with Parallel LINQ 353
 - Plotting the Mandelbrot set with a single thread* 353
 - *Introducing ParallelEnumerable, ParallelQuery, and AsParallel* 354
 - Tweaking parallel queries* 356

- 12.5 Inverting the query model with LINQ to Rx 357
 - IObservable<T> and IObservable<T> 358* ■ *Starting simply (again) 360* ■ *Querying observables 360* ■ *What's the point? 363*
- 12.6 Extending LINQ to Objects 364
 - Design and implementation guidelines 364* ■ *Sample extension: selecting a random element 365*
- 12.7 Summary 367

PART 4 C# 4: PLAYING NICELY WITH OTHERS369

13 *Minor changes to simplify code* 371

- 13.1 Optional parameters and named arguments 372
 - Optional parameters 372* ■ *Named arguments 378* ■ *Putting the two together 382*
- 13.2 Improvements for COM interoperability 387
 - The horrors of automating Word before C# 4 387* ■ *The revenge of optional parameters and named arguments 388* ■ *When is a ref parameter not a ref parameter? 389* ■ *Calling named indexers 390* ■ *Linking primary interop assemblies 391*
- 13.3 Generic variance for interfaces and delegates 394
 - Types of variance: covariance and contravariance 394* ■ *Using variance in interfaces 396* ■ *Using variance in delegates 399* ■ *Complex situations 399* ■ *Restrictions and notes 401*
- 13.4 Teeny tiny changes to locking and field-like events 405
 - Robust locking 405* ■ *Changes to field-like events 406*
- 13.5 Summary 407

14 *Dynamic binding in a static language* 409

- 14.1 What? When? Why? How? 411
 - What is dynamic typing? 411* ■ *When is dynamic typing useful, and why? 412* ■ *How does C# 4 provide dynamic typing? 413*
- 14.2 The five-minute guide to dynamic 414
- 14.3 Examples of dynamic typing 416
 - COM in general, and Microsoft Office in particular 417* ■ *Dynamic languages such as IronPython 419* ■ *Dynamic typing in purely managed code 423*

- 14.4 Looking behind the scenes 429
 - Introducing the Dynamic Language Runtime* 429 ■ *DLR core concepts* 431 ■ *How the C# compiler handles dynamic* 434
 - The C# compiler gets even smarter* 438 ■ *Restrictions on dynamic code* 441
- 14.5 Implementing dynamic behavior 444
 - Using ExpandableObject* 444 ■ *Using DynamicObject* 448
 - Implementing IDynamicMetaObjectProvider* 455
- 14.6 Summary 459

PART 5 C# 5: ASYNCHRONY MADE SIMPLE461

15 *Asynchrony with async/await* 463

- 15.1 Introducing asynchronous functions 465
 - First encounters of the asynchronous kind* 465 ■ *Breaking down the first example* 467
- 15.2 Thinking about asynchrony 468
 - Fundamentals of asynchronous execution* 468 ■ *Modeling asynchronous methods* 471
- 15.3 Syntax and semantics 472
 - Declaring an async method* 472 ■ *Return types from async methods* 473 ■ *The awaitable pattern* 474 ■ *The flow of await expressions* 477 ■ *Returning from an async method* 481
 - Exceptions* 482
- 15.4 Asynchronous anonymous functions 490
- 15.5 Implementation details: compiler transformation 492
 - Overview of the generated code* 493 ■ *Structure of the skeleton method* 495 ■ *Structure of the state machine* 497 ■ *One entry point to rule them all* 498 ■ *Control around await expressions* 500 ■ *Keeping track of a stack* 501 ■ *Finding out more* 503
- 15.6 Using async/await effectively 503
 - The task-based asynchronous pattern* 504 ■ *Composing async operations* 507 ■ *Unit testing asynchronous code* 511
 - The awaitable pattern redux* 515 ■ *Asynchronous operations in WinRT* 516
- 15.7 Summary 517

16	C# 5 bonus features and closing thoughts	519
16.1	Changes to captured variables in foreach loops	520
16.2	Caller information attributes	520
	<i>Basic behavior</i>	521
	<i>Logging</i>	522
	<i>Implementing</i>	
	<i>INotifyPropertyChanged</i>	523
	<i>Using caller information attributes</i>	
	<i>without .NET 4.5</i>	524
16.3	Closing thoughts	525
<i>appendix A</i>	<i>LINQ standard query operators</i>	<i>527</i>
<i>appendix B</i>	<i>Generic collections in .NET</i>	<i>540</i>
<i>appendix C</i>	<i>Version summaries</i>	<i>554</i>
	<i>index</i>	<i>563</i>

foreword

There are two kinds of pianists.

There are some pianists who play, not because they enjoy it, but because their parents force them to take lessons. Then there are those who play the piano because it pleases them to create music. They don't need to be forced; on the contrary, they sometimes don't know when to stop.

Of the latter kind, there are some who play the piano as a hobby. Then there are those who play for a living. That requires a higher level of dedication, skill, and talent. They may have some degree of freedom about what genre of music they play and the stylistic choices they make in playing it, but fundamentally those choices are driven by the needs of the employer or the tastes of the audience.

Of the latter kind, there are some who do it primarily for the money. Then there are those professionals who would want to play the piano in public even if they weren't being paid. They enjoy using their skills and talents to make music for others. That they can have fun and get paid for it is so much the better.

Of the latter kind, there are some who are self-taught, who play by ear, who might have great talent and ability, but can't communicate that intuitive understanding to others except through the music itself. Then there are those who have formal training in both theory and practice. They can explain what techniques the composer used to achieve the intended emotional effect, and use that knowledge to shape their interpretation of the piece.

Of the latter kind, there are some who have never looked inside their pianos. Then there are those who are fascinated by the clever escapements that lift the damper felts a fraction of a second before the hammers strike the strings. They own key levelers

and capstan wrenches. They take delight and pride in being able to understand the mechanisms of an instrument that has 5–10,000 moving parts.

Of the latter kind, there are some who are content to master their craft and exercise their talents for the pleasure and profit it brings. Then there are those who are not just artists, theorists, and technicians; somehow they find the time to pass that knowledge on to others as mentors.

I have no idea if Jon Skeet is a pianist or musician of any sort. But from my email conversations with him as one of the C# team's Most Valuable Professionals over the years, from reading his blog, and from reading every word of each of his books at least three times, it has become clear to me that Jon is that latter kind of software developer: enthusiastic, knowledgeable, talented, curious, analytical—and a teacher of others.

C# is a highly pragmatic and rapidly evolving language. Through the addition of query comprehensions, richer type inference, a compact syntax for anonymous functions, and so on, I hope that we have enabled a whole new style of programming while still staying true to the statically typed, component-oriented approach that has made C# a success.

Many of these new stylistic elements have the paradoxical quality of feeling very old (lambda expressions go back to the foundations of computer science in the first half of the twentieth century) and yet at the same time feeling new and unfamiliar to developers used to a more modern object-oriented approach.

Jon gets all that. This book is ideal for professional developers who have a need to understand the *what* and *how* of the latest revision to C#. But it is also for those developers whose understanding is enriched by exploring the *why* of the language's design principles.

Being able to take advantage of all that new power requires new ways of thinking about data, functions, and the relationship between them. It's not unlike trying to play jazz after years of classical training—or vice versa. Either way, I'm looking forward to finding out what sorts of functional compositions the next generation of C# programmers come up with. Happy composing, and thanks for choosing the key of C# to do it in.

ERIC LIPPERT
C# ANALYSIS ARCHITECT
COVERITY

preface

Oh boy. When writing this preface, I started off with the preface to the second edition, which began by saying how long it felt since writing the preface to the first edition. The second edition is now a distant memory, and the first edition seems like a whole different life. I'm not sure whether that says more about the pace of modern life or my memory, but it's a sobering thought either way.

The development landscape has changed enormously since the first edition, and even since the second. This has been driven by many factors, with the rise of mobile devices probably being the most obvious. But many challenges have remained the same. It's still hard to write properly internationalized applications. It's still hard to handle errors gracefully in all situations. It's still fairly hard to write correct multi-threaded applications, although this task has been made significantly simpler by both language and library improvements over the years.

Most importantly in the context of this preface, I believe developers still need to know the language they're using at a level where they're confident in how it will behave. They may not know the fine details of every API call they're using, or even some of the obscure corner cases of the language that they don't happen to use,¹ but the core of the language should feel like a solid friend that the developer can rely on to behave predictably.

In addition to the letter of the language you're developing in, I believe there's great benefit in understanding its spirit. While you may occasionally find you have a fight on your hands however hard you try, if you attempt to make your code work in the way the language designers intended, your experience will be a much more pleasant one.

¹ I have a confession to make: I know very little about unsafe code and pointers in C#. I've simply never needed to find out about them.

acknowledgments

You might expect that putting together a third edition—and one where the main change consists of two new chapters—would be straightforward. Indeed, writing the “green field” content of chapters 15 and 16 was the easy part. But there’s a lot more to it than that—tweaking little bits of language throughout the rest of the book, checking for any aspects which were fine a few years ago but don’t quite make sense now, and generally making sure the whole book is up to the high standards I expect readers to hold it to. Fortunately, I have been lucky enough to have a great set of people supporting me and keeping the book on the straight and narrow.

Most importantly, my family have been as wonderful as ever. My wife Holly is a children’s author herself, so our kids are used to us having to lock ourselves away for a while to meet editorial deadlines, but they’ve remained cheerfully encouraging throughout. Holly herself takes all of this in stride, and I’m grateful that she’s never reminded me just how many books she’s started from scratch and completed in the time I’ve been working on this third edition.

The formal peer reviewers are listed later on, but I’d like to add a note of personal thanks to all those who ordered early access copies of this third edition, finding typos and suggesting changes...also constantly asking when the book was coming out. The very fact that I had readers who were eager to get their hands on the finished book was a huge source of encouragement.

I always get on well with the team at Manning, and it’s been a pleasure to work with some familiar friends from the first edition as well as newcomers. Mike Stephens and Jeff Bleiel have guided the whole process smoothly, as we decided what to change from the earlier editions and what to keep. They’ve generally put the whole thing into

the right shape. Andy Carroll and Katie Tennant provided expert copyediting and proofreading, respectively, never once expressing irritation with my Englishness, pickiness, or general bewilderment. The production team has worked its magic in the background, as ever, but I'm grateful to them nonetheless: Dottie Marsico, Janet Vail, Marija Tudor, and Mary Piergies. Finally, I'd like to thank the publisher, Marjan Bace, for allowing me a third edition and exploring some interesting future options.

Peer review is immensely important, not only for getting the technical details of the book right, but also the balance and tone. Sometimes the comments we received have merely shaped the overall book; in other cases I've made very specific changes in response. Either way, all feedback has been welcome. So thanks to the following reviewers for making the book better for all of us: Andy Kirsch, Bas Pennings, Bret Colloff, Charles M. Gross, Dror Helper, Dustin Laine, Ivan Todorović, Jon Parish, Sebastian Martín Aguilar, Tiaan Geldenhuys, and Timo Bredenoort.

I'd particularly like to thank Stephen Toub and Stephen Cleary, whose early reviews of chapter 15 were invaluable. Asynchrony is a particularly tricky topic to write about clearly but accurately, and their expert advice made a very significant difference to the chapter.

Without the C# team, this book would have no cause to exist, of course. Their dedication to the language in design, implementation and testing is exemplary, and I look forward to seeing what they come up with next. Since the second edition was published, Eric Lippert has left the C# team for a new fabulous adventure, but I'm enormously grateful that he was still able to act as the tech reviewer for this third edition. I also thank him for the foreword that he originally wrote to the first edition and that is included again this time. I refer to Eric's thoughts on various matters throughout the book, and if you aren't already reading his blog (<http://ericlippert.com>), you really should be.

about this book

This is a book about C# from version 2 onward—it's as simple as that. I barely cover C# 1 and only cover the .NET Framework libraries and Common Language Runtime (CLR) when they're related to the language. This is a deliberate decision, and the result is a book quite different from most of the C# and .NET books I've seen.

By assuming a reasonable amount of knowledge of C# 1, I avoid spending hundreds of pages covering material that I think most people already understand. This gives me room to expand on the details of later versions of C#, which is what I hope you're reading the book for. When I wrote the first edition of this book, even C# 2 was relatively unknown to some readers. By now, almost all C# developers have some experience with the features introduced in C# 2, but I've still kept that material in this edition, as it's so fundamental to what comes later.

Who should read this book?

This book is squarely aimed at developers who already know some C#. For absolute maximum value, you'd know C# 1 well but know very little about later versions. There aren't many readers in that sweet spot any more, but I believe there are still lots of developers who can benefit from digging deeper into C# 2 and 3, even if they've already been using them for a while...and many developers haven't yet used C# 4 or 5 to any extent.

If you don't know any C# at all, this probably isn't the book for you. You could struggle through, looking up aspects you're not familiar with, but it wouldn't be a very efficient way of learning. You'd be better off starting with a different book, and then gradually adding *C# in Depth* to the mix. There's a wide variety of books that cover C#

from scratch, in many different styles. The *C# in a Nutshell* series (O'Reilly) has always been good in this respect, and *Essential C# 5.0* (Addison-Wesley Professional) is also a good introduction.

I'm not going to claim that reading this book will make you a fabulous coder. There's so much more to software engineering than knowing the syntax of the language you happen to be using. I give some words of guidance, but ultimately there's a lot more gut instinct in development than most of us would like to admit. What I will claim is that if you read and understand this book, you should feel comfortable with C# and free to follow your instincts without too much apprehension. It's not about being able to write code that no one else will understand because it uses unknown corners of the language; it's about being confident that you know the options available to you, and know which path the C# idioms are encouraging you to follow.

Roadmap

The book's structure is simple. There are five parts and three appendixes. The first part serves as an introduction, including a refresher on topics in C# 1 that are important for understanding later versions of the language, and that are often misunderstood. The second part covers the new features introduced in C# 2, the third part covers C# 3, and so on.

There are occasions when organizing the material this way means we'll come back to a topic a couple of times—in particular, delegates are improved in C# 2 and then again in C# 3—but there is method in my madness. I anticipate that a number of readers will be using different versions for different projects; for example, you may be using C# 4 at work, but experimenting with C# 5 at home. That means it's useful to clarify what is in which version. It also provides a feeling of context and evolution—it shows how the language has developed over time.

Chapter 1 sets the scene by taking a simple piece of C# 1 code and evolving it, seeing how later versions allow the source to become more readable and powerful. We'll look at the historical context in which C# has grown, and the technical context in which it operates as part of a complete platform; C# as a language builds on framework libraries and a powerful runtime to turn abstraction into reality.

Chapter 2 looks back at C# 1, and at three specific aspects: delegates, the type system characteristics, and the differences between value types and reference types. These topics are often understood “just well enough” by C# 1 developers, but as C# has evolved and developed them significantly, a solid grounding is required in order to make the most of the new features.

Chapter 3 tackles the biggest feature of C# 2, and potentially the hardest to grasp: generics. Methods and types can be written generically, with type parameters standing in for real types that are specified in the calling code. Initially it's as confusing as this description makes it sound, but once you understand generics, you'll wonder how you survived without them.

If you've ever wanted to represent a null integer, chapter 4 is for you. It introduces nullable types: a feature, built on generics, that takes advantage of support in the language, runtime, and framework.

Chapter 5 shows the improvements to delegates in C# 2. Until now, you may have only used delegates for handling events such as button clicks. C# 2 makes it easier to create delegates, and library support makes them more useful for situations other than events.

In chapter 6 we'll examine iterators, and the easy way to implement them in C# 2. Few developers use iterator blocks, but as LINQ to Objects is built on iterators, they'll become more and more important. The lazy nature of their execution is also a key part of LINQ.

Chapter 7 shows a number of smaller features introduced in C# 2, each making life a little more pleasant. The language designers have smoothed over a few rough places in C# 1, allowing more flexible interaction with code generators, better support for utility classes, more granular access to properties, and more.

Chapter 8 once again looks at a few relatively simple features—but this time in C# 3. Almost all the new syntax is geared toward the common goal of LINQ, but the building blocks are also useful in their own right. With anonymous types, automatically implemented properties, implicitly typed local variables, and greatly enhanced initialization support, C# 3 gives a far richer language with which your code can express its behavior.

Chapter 9 looks at the first major topic of C# 3—lambda expressions. Not content with the reasonably concise syntax discussed in chapter 5, the language designers have made delegates even easier to create than in C# 2. Lambdas are capable of more—they can be converted into expression trees, a powerful way of representing code as data.

In chapter 10 we'll examine extension methods, which provide a way of fooling the compiler into believing that methods declared in one type actually belong to another. At first glance this appears to be a readability nightmare, but with careful consideration it can be an extremely powerful feature—and one that's vital to LINQ.

Chapter 11 combines the previous three chapters in the form of query expressions, a concise but powerful way of querying data. Initially we'll concentrate on LINQ to Objects, but you'll see how the query expression pattern is applied in a way that allows other data providers to plug in seamlessly.

Chapter 12 is a quick tour of various different uses of LINQ. First we'll look at the benefits of query expressions combined with expression trees—how LINQ to SQL is able to convert what appears to be normal C# into SQL statements. We'll then move on to see how libraries can be designed to mesh well with LINQ, taking LINQ to XML as an example. Parallel LINQ and Reactive Extensions show two alternative approaches to in-process querying, and the chapter closes with a discussion of how you can extend LINQ to Objects with your own LINQ operators.

Coverage of C# 4 begins in chapter 13, where we'll look at named arguments and optional parameters, COM interop improvements, and generic variance. In some ways these are very separate features, but named arguments and optional parameters contribute to COM interop as well as the more specific abilities that are only available when working with COM objects.

Chapter 14 describes the single biggest feature in C# 4: dynamic typing. The ability to bind members dynamically at execution time instead of statically at compile time is a huge departure for C#, but it's applied selectively—only code that involves a dynamic value will be executed dynamically.

Chapter 15 is all about asynchrony. C# 5 only contains one major feature—the ability to write asynchronous functions. This single feature is simultaneously brain-bustingly complicated to understand thoroughly and awe-inspiringly elegant to use. At long last, we can write asynchronous code that doesn't read like spaghetti.

We'll wind down in chapter 16 with the remaining features of C# 5 (both of which are tiny) and some thoughts about the future.

The appendixes are all reference material. In appendix A, I cover the LINQ standard query operators, with some examples. Appendix B looks at the core generic collection classes and interfaces. Appendix C provides a brief look at the different versions of .NET, including the different flavors such as the Compact Framework and Silverlight.

Terminology, typography, and downloads

Most of the terminology of the book is explained as it goes along, but there are a few definitions that are worth highlighting here. I use C# 1, C# 2, C# 3, C# 4, and C# 5 in a reasonably obvious manner—but you may see other books and websites referring to C# 1.0, C# 2.0, C# 3.0, C# 4.0, and C# 5.0. The extra “.0” seems redundant to me, which is why I've omitted it—I hope the meaning is clear.

I've appropriated a pair of terms from a C# book by Mark Michaelis. To avoid the confusion between *runtime* being an execution environment (as in “the Common Language Runtime”) and a point in time (as in “overriding occurs at runtime”), Mark uses *execution time* for the latter concept, usually in comparison with *compile time*. This seems to me to be a thoroughly sensible idea, and one that I hope catches on in the wider community. I'm doing my bit by following his example in this book.

I frequently refer to “the language specification” or just “the specification”—unless I indicate otherwise, this means the C# language specification. However, multiple versions of the specification are available, partly due to different versions of the language itself and partly due to the standardization process. Any section numbers provided are from the C# 5.0 language specification from Microsoft.

This book contains numerous pieces of code, which appear in a fixed-width font like this; output from the listings appears in the same way. Code annotations accompany some listings, and at other times particular sections of the code are shown in bold to highlight a change, improvement, or addition. Almost all of the code

appears in snippet form, allowing it to stay compact but still runnable—within the right environment. That environment is Snippy, a custom tool that is introduced in section 1.8. Snippy is available for download, along with all of the code from the book (in the form of snippets, full Visual Studio solutions, or more often both) from the book’s website at csharpindepth.com, as well as from the publisher's website at manning.com/CSharpinDepthThirdEdition.

Author Online and the C# in Depth website

Purchase of *C# in Depth, Third Edition* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and other users. To access the forum and subscribe to it, point your web browser to www.manning.com/CSharpinDepthThirdEdition. This page provides information on how to get on the forum once you are registered, what kind of help is available, and the rules of conduct on the forum.

The Author Online forum and the archives of previous discussions will be accessible from the publisher’s website as long as the book is in print.

In addition to Manning’s own website, I have set up a companion website for the book at csharpindepth.com, containing information that didn’t quite fit into the book, downloadable source code for all the listings in the book, and links to other resources.

about the author

I'm not a typical C# developer, I think it's fair to say. For the last five years, almost all of my time working with C# has been for fun—effectively as a somewhat obsessive hobby. At work, I've been writing server-side Java in Google London, and I can safely claim that few things help you to appreciate new language features more than having to code in a language that doesn't have them, but is similar enough to remind you of their absence.

I've tried to keep in touch with what other developers find hard about C# by keeping a careful eye on Stack Overflow, posting oddities to my blog, and occasionally talking about C# and related topics just about anywhere that will provide people to listen to me. Additionally, I'm actively developing an open source .NET date and time API called Noda Time (see <http://nodatime.org>). In short, C# is still coursing through my veins as strongly as ever.

For all these oddities—and despite my ever-surprising micro-celebrity status due to Stack Overflow—I'm a very ordinary developer in many other ways. I write plenty of code that makes me grimace when I come back to it. My unit tests don't always come first...and sometimes they don't even exist. I make off-by-one errors every so often. The type inference section of the C# specification still confuses me, and there are some uses of Java wildcards that make me want to have a little lie-down. I'm a deeply flawed programmer.

That's the way it should be. For the next few hundred pages, I'll try to pretend otherwise: I'll espouse best practices as if I always followed them myself, and frown on dirty shortcuts as if I'd never dream of taking them. Don't believe a word of it. The truth of the matter is, I'm probably just like you. I happen to know a bit more about how C# works, that's all...and even that state of affairs will only last until you've finished the book.

about the cover illustration

The caption for the illustration on the cover of *C# in Depth, Third Edition* is “Musician.” The illustration is taken from a collection of costumes of the Ottoman Empire published on January 1, 1802, by William Miller of Old Bond Street, London. The title page is missing from the collection and we have been unable to track it down to date. The book’s table of contents identifies the figures in both English and French, and each illustration bears the names of two artists who worked on it, both of whom would no doubt be surprised to find their art gracing the front cover of a computer programming book...two hundred years later.

The collection was purchased by a Manning editor at an antiquarian flea market in the “Garage” on West 26th Street in Manhattan. The seller was an American based in Ankara, Turkey, and the transaction took place just as he was packing up his stand for the day. The Manning editor didn't have on his person the substantial amount of cash that was required for the purchase and a credit card and check were both politely turned down. With the seller flying back to Ankara that evening, the situation was getting hopeless. What was the solution? It turned out to be nothing more than an old-fashioned verbal agreement sealed with a handshake. The seller simply proposed that the money be transferred to him by wire and the editor walked out with the bank information on a piece of paper and the portfolio of images under his arm. Needless to say, we transferred the funds the next day, and we remain grateful and impressed by this unknown person’s trust in one of us. It recalls something that might have happened a long time ago.

We at Manning celebrate the inventiveness, the initiative, and, yes, the fun of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by the pictures from this collection.

Part 1

Preparing for the journey

Every reader will come to this book with a different set of expectations and a different level of experience. Are you an expert looking to fill some holes, however small, in your present knowledge? Perhaps you consider yourself an average developer, with a bit of experience in using generics and lambda expressions, but a desire to better understand how they work. Maybe you're reasonably confident with C# 2 and 3 but have no experience with C# 4 or 5.

As an author, I can't make every reader the same—and I wouldn't want to, even if I could. But I hope that all readers have two things in common: the desire for a deeper relationship with C# as a language, and at least a basic knowledge of C# 1. If you can bring those elements to the party, I'll provide the rest.

The potentially huge range of skill levels is the main reason why this part of the book exists. You may already know what to expect from later versions of C#—or it could all be brand new to you. You could have a rock-solid understanding of C# 1, or you might be rusty on some of the details—some of which will become increasingly important as you learn about the later versions. By the end of part 1, I won't have leveled the playing field entirely, but you should be able to approach the rest of the book with confidence and an idea of what's coming later.

In the first two chapters, we'll look both forward and back. One of the key themes of the book is evolution. Before introducing any feature into the language, the C# design team carefully considers that feature in the context of what's already present and the general goals for the future. This brings a feeling of consistency to the language even in the midst of change. To understand how and why the language is evolving, you need to see where it's come from and where it's going.

Chapter 1 presents a bird's-eye view of the rest of the book, taking a brief look at some of the biggest features of C# beyond version 1. I'll show a progression of code from C# 1 onward, applying new features one by one until the code is almost unrecognizable from its humble beginnings. We'll also look at some of the terminology I'll use in the rest of the book, as well as the format for the sample code.

Chapter 2 is heavily focused on C# 1. If you're an expert in C# 1, you can skip this chapter, but it does tackle some of the areas of C# 1 that tend to be misunderstood. Rather than try to explain the whole of the language, the chapter concentrates on features that are fundamental to the later versions of C#. From this solid base, you can move on and look at C# 2 in part 2 of the book.

The changing face of C# development



This chapter covers

- An evolving example
- The composition of .NET
- Using the code in this book
- The C# language specification

Do you know what I really like about dynamic languages such as Python, Ruby, and Groovy? They suck away fluff from your code, leaving just the essence of it—the bits that really *do* something. Tedious formality gives way to features such as generators, lambda expressions, and list comprehensions.

The interesting thing is that few of the features that tend to give dynamic languages their lightweight feel have anything to do with being dynamic. Some do, of course—duck typing and some of the magic used in Active Record, for example—but statically typed languages don't *have* to be clumsy and heavyweight.

Enter C#. In some ways, C# 1 could have been seen as a nicer version of the Java language, circa 2001. The similarities were all too clear, but C# had a few extras: properties as a first-class feature in the language, delegates and events, foreach

loops, using statements, explicit method overriding, operator overloading, and custom value types, to name a few. Obviously, language preference is a personal issue, but C# 1 definitely felt like a step up from Java when I first started using it.

Since then, things have only gotten better. Each new version of C# has added significant features to reduce developer angst, but always in a carefully considered way, and with little backward incompatibility. Even before C# 4 gained the ability to use dynamic typing where it's genuinely useful, many features traditionally associated with dynamic and functional languages had made it into C#, leading to code that's easier to write and maintain. Similarly, while the features around asynchrony in C# 5 aren't exactly the same as those in F#, it feels to me like there's a definite influence.

In this book, I'll take you through those changes one by one, in enough detail to make you feel comfortable with some of the miracles the C# compiler is now prepared to perform on your behalf. All that comes later, though—in this chapter I'll whiz through as many features as I can, barely taking a breath. I'll define what I mean when I talk about C# as a language compared with .NET as a platform, and I'll offer a few important notes about the sample code for the rest of the book. Then we can dive into the details.

We won't be looking at *all* the changes made to C# in this single chapter, but you'll see generics, properties with different access modifiers, nullable types, anonymous methods, automatically implemented properties, enhanced collection initializers, enhanced object initializers, lambda expressions, extension methods, implicit typing, LINQ query expressions, named arguments, optional parameters, simpler COM interop, dynamic typing, and asynchronous functions. These will carry us from C# 1 all the way up to the latest release, C# 5. Obviously that's a lot to get through, so let's get started.

1.1 **Starting with a simple data type**

In this chapter I'll let the C# compiler do amazing things without telling you how and barely mentioning the what or the why. This is the only time that I won't explain how things work or try to go one step at a time. Quite the opposite, in fact—the plan is to impress rather than educate. If you read this entire section without getting at least a little excited about what C# can do, maybe this book isn't for you. With any luck, though, you'll be eager to get to the details of how these magic tricks work, and that's what the rest of the book is for.

The example I'll use is contrived—it's designed to pack as many new features into as short a piece of code as possible. It's also clichéd, but at least that makes it familiar. Yes, it's a product/name/price example, the e-commerce alternative to “hello, world.” We'll look at how various tasks can be achieved, and how, as we move forward in versions of C#, you can accomplish them more simply and elegantly than before. You won't see any of the benefits of C# 5 until right at the end, but don't worry—that doesn't make it any less important.

1.1.1 The Product type in C# 1

We'll start off with a type representing a product, and then manipulate it. You won't see anything particularly impressive yet—just the encapsulation of a couple of properties. To make life simpler for demonstration purposes, this is also where we'll create a list of predefined products.

Listing 1.1 shows the type as it might be written in C# 1. We'll then move on to see how the code might be rewritten for each later version. This is the pattern we'll follow for each of the other pieces of code. Given that I'm writing this in 2013, it's likely that you're already familiar with code that uses some of the features I'll introduce, but it's worth looking back so you can see how far the language has come.

Listing 1.1 The Product type (C# 1)

```
using System.Collections;
public class Product
{
    string name;
    public string Name { get { return name; } }

    decimal price;
    public decimal Price { get { return price; } }

    public Product(string name, decimal price)
    {
        this.name = name;
        this.price = price;
    }

    public static ArrayList GetSampleProducts()
    {
        ArrayList list = new ArrayList();
        list.Add(new Product("West Side Story", 9.99m));
        list.Add(new Product("Assassins", 14.99m));
        list.Add(new Product("Frogs", 13.99m));
        list.Add(new Product("Sweeney Todd", 10.99m));
        return list;
    }

    public override string ToString()
    {
        return string.Format("{0}: {1}", name, price);
    }
}
```

Nothing in listing 1.1 should be hard to understand—it's just C# 1 code, after all. There are three limitations that it demonstrates, though:

- An `ArrayList` has no compile-time information about what's in it. You could accidentally add a string to the list created in `GetSampleProducts`, and the compiler wouldn't bat an eyelid.
- You've provided public getter properties, which means that if you wanted matching setters, they'd have to be public, too.

- There's a lot of fluff involved in creating the properties and variables—code that complicates the simple task of encapsulating a string and a decimal.

Let's see what C# 2 can do to improve matters.

1.1.2 **Strongly typed collections in C# 2**

Our first set of changes (shown in the following listing) tackles the first two items listed previously, including the most important change in C# 2: generics. The parts that are new are in bold.

Listing 1.2 Strongly typed collections and private setters (C# 2)

```
public class Product
{
    string name;
    public string Name
    {
        get { return name; }
        private set { name = value; }
    }

    decimal price;
    public decimal Price
    {
        get { return price; }
        private set { price = value; }
    }

    public Product(string name, decimal price)
    {
        Name = name;
        Price = price;
    }

    public static List<Product> GetSampleProducts()
    {
        List<Product> list = new List<Product>();
        list.Add(new Product("West Side Story", 9.99m));
        list.Add(new Product("Assassins", 14.99m));
        list.Add(new Product("Frogs", 13.99m));
        list.Add(new Product("Sweeney Todd", 10.99m));
        return list;
    }

    public override string ToString()
    {
        return string.Format("{0}: {1}", name, price);
    }
}
```

You now have properties with private setters (which you use in the constructor), and it doesn't take a genius to guess that `List<Product>` is telling the compiler that the list contains products. Attempting to add a different type to the list would result in a compiler error, and you also don't need to cast the results when you fetch them from the list.

The changes in C# 2 leave only one of the original three difficulties unanswered, and C# 3 helps out there.

1.1.3 Automatically implemented properties in C# 3

We're starting off with some fairly tame features from C# 3. The automatically implemented properties and simplified initialization shown in the following listing are relatively trivial compared with lambda expressions and the like, but they can make code a lot simpler.

Listing 1.3 Automatically implemented properties and simpler initialization (C# 3)

```
using System.Collections.Generic;

class Product
{
    public string Name { get; private set; }
    public decimal Price { get; private set; }

    public Product(string name, decimal price)
    {
        Name = name;
        Price = price;
    }

    Product() {}

    public static List<Product> GetSampleProducts()
    {
        return new List<Product>
        {
            new Product { Name="West Side Story", Price = 9.99m },
            new Product { Name="Assassins", Price=14.99m },
            new Product { Name="Frogs", Price=13.99m },
            new Product { Name="Sweeney Todd", Price=10.99m }
        };
    }

    public override string ToString()
    {
        return string.Format("{0}: {1}", Name, Price);
    }
}
```

Now the properties don't have any code (or visible variables!) associated with them, and you're building the hardcoded list in a very different way. With no name and price variables to access, you're forced to use the properties everywhere in the class, improving consistency. You now have a private parameterless constructor for the sake of the new property-based initialization. (This constructor is called for each item before the properties are set.)

In this example, you could've removed the public constructor completely, but then no outside code could've created other product instances.

1.1.4 Named arguments in C# 4

For C# 4, we'll go back to the original code when it comes to the properties and constructor, so that it's fully immutable again. A type with only private setters can't be *publicly* mutated, but it can be clearer if it's not privately mutable either.¹ There's no shortcut for read-only properties, unfortunately, but C# 4 lets you specify argument names for the constructor call, as shown in the following listing, which gives you the clarity of C# 3 initializers without the mutability.

Listing 1.4 Named arguments for clear initialization code (C# 4)

```
using System.Collections.Generic;
public class Product
{
    readonly string name;
    public string Name { get { return name; } }

    readonly decimal price;
    public decimal Price { get { return price; } }

    public Product(string name, decimal price)
    {
        this.name = name;
        this.price = price;
    }

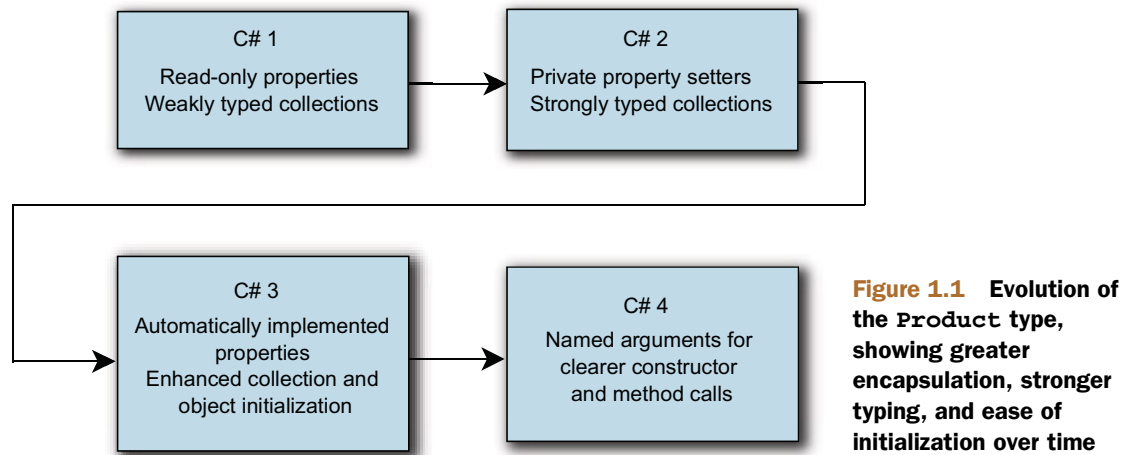
    public static List<Product> GetSampleProducts()
    {
        return new List<Product>
        {
            new Product( name: "West Side Story", price: 9.99m),
            new Product( name: "Assassins", price: 14.99m),
            new Product( name: "Frogs", price: 13.99m),
            new Product( name: "Sweeney Todd", price: 10.99m)
        };
    }

    public override string ToString()
    {
        return string.Format("{0}: {1}", name, price);
    }
}
```

The benefits of specifying the argument names explicitly are relatively minimal in this particular example, but when a method or constructor has several parameters, it can make the meaning of the code much clearer—particularly if they're of the same type, or if you're passing in null for some arguments. You can choose when to use this feature, of course, only specifying the names for arguments when it makes the code easier to understand.

Figure 1.1 summarizes how the `Product` type has evolved so far. I'll include a similar diagram after each task, so you can see the pattern of how the evolution of C#

¹ The C# 1 code could've been immutable too—I only left it mutable to simplify the changes for C# 2 and 3.



improves the code. You'll notice that C# 5 is missing from all of the block diagrams; that's because the main feature of C# 5 (asynchronous functions) is aimed at an area that really hasn't evolved much in terms of language support. We'll take a peek at it before too long, though.

So far, the changes are relatively minimal. In fact, the addition of generics (the `List<Product>` syntax) is probably the most important part of C# 2, but you've only seen part of its usefulness so far. There's nothing to get the heart racing yet, but we've only just started. Our next task is to print out the list of products in alphabetical order.

1.2 Sorting and filtering

In this section, we won't change the `Product` type at all—instead, we'll take the sample products and sort them by name, and then find the expensive ones. Neither of these tasks is exactly *difficult*, but you'll see how much simpler they become over time.

1.2.1 Sorting products by name

The easiest way to display a list in a particular order is to sort the list and then run through it, displaying items. In .NET 1.1, this involved using `ArrayList.Sort`, and optionally providing an `IComparer` implementation to specify a particular comparison. You could make the `Product` type implement `IComparable`, but that would only allow you to define one sort order, and it's not a stretch to imagine that you might want to sort by price at some stage, as well as by name.

The following listing implements `IComparer`, and then sorts the list and displays it.

Listing 1.5 Sorting an `ArrayList` using `IComparer` (C# 1)

```

class ProductNameComparer : IComparer
{
    public int Compare(object x, object y)
    {
        Product first = (Product)x;
        Product second = (Product)y;
        return first.Name.CompareTo(second.Name);
    }
}
  
```



```

    }
}
...
ArrayList products = Product.GetSampleProducts();
products.Sort(new ProductNameComparer());
foreach (Product product in products)
{
    Console.WriteLine (product);
}

```

The first thing to spot in listing 1.5 is that you had to introduce an extra type to help with the sorting. That's not a disaster, but it's a lot of code if you only want to sort by name in one place. Next, look at the casts in the `Compare` method. Casts are a way of telling the compiler that you know more information than it does, and that usually means there's a chance you're wrong. If the `ArrayList` you returned from `GetSampleProducts` *did* contain a string, that's where the code would go bang—where the comparison tries to cast the string to a `Product`.

You also have a cast in the code that displays the sorted list. It's not obvious, because the compiler puts it in automatically, but the `foreach` loop implicitly casts each element of the list to `Product`. Again, that cast could fail at execution time, and once more generics come to the rescue in C# 2. The following listing shows the previous code with the use of generics as the *only* change.

Listing 1.6 Sorting a `List<Product>` using `IComparer<Product>` (C# 2)

```

class ProductNameComparer : IComparer<Product>
{
    public int Compare(Product x, Product y)
    {
        return x.Name.CompareTo(y.Name);
    }
}
...
List<Product> products = Product.GetSampleProducts();
products.Sort(new ProductNameComparer());
foreach (Product product in products)
{
    Console.WriteLine(product);
}

```

The code for the comparer in listing 1.6 is simpler because you're given products to start with. No casting is necessary. Similarly, the invisible cast in the `foreach` loop is effectively gone now. The compiler still has to consider the conversion from the source type of the sequence to the target type of the variable, but it knows that in this case both types are `Product`, so it doesn't need to emit any code for the conversion.

That's an improvement, but it'd be nice if you could sort the products by simply specifying the comparison to make, without needing to implement an interface to do so. The following listing shows how to do precisely this, telling the `Sort` method how to compare two products using a delegate.

Listing 1.7 Sorting a `List<Product>` using `Comparison<Product>` (C# 2)

```
List<Product> products = Product.GetSampleProducts();  
  
products.Sort(delegate(Product x, Product y)  
    { return x.Name.CompareTo(y.Name); }  
);  
foreach (Product product in products)  
{  
    Console.WriteLine(product);  
}
```

Behold the lack of the `ProductNameComparer` type. The statement in bold font creates a delegate instance, which you provide to the `Sort` method in order to perform the comparisons. You'll learn more about this feature (*anonymous methods*) in chapter 5.

You've now fixed all the problems identified in the C# 1 version. That doesn't mean that C# 3 can't do better, though. First, you'll replace the anonymous method with an even more compact way of creating a delegate instance, as shown in the following listing.

Listing 1.8 Sorting using `Comparison<Product>` from a lambda expression (C# 3)

```
List<Product> products = Product.GetSampleProducts();  
products.Sort((x, y) => x.Name.CompareTo(y.Name));  
foreach (Product product in products)  
{  
    Console.WriteLine(product);  
}
```

You've gained even more strange syntax (a *lambda expression*), which still creates a `Comparison<Product>` delegate, just as listing 1.7 did, but this time with less fuss. You didn't have to use the `delegate` keyword to introduce it, or even specify the types of the parameters.

There's more, though: with C# 3, you can easily print out the names in order without modifying the original list of products. The next listing shows this using the `OrderBy` method.

Listing 1.9 Ordering a `List<Product>` using an extension method (C# 3)

```
List<Product> products = Product.GetSampleProducts();  
foreach (Product product in products.OrderBy(p => p.Name) )  
{  
    Console.WriteLine (product);  
}
```

In this listing, you appear to be calling an `OrderBy` method on the list, but if you look in MSDN, you'll see that it doesn't even exist in `List<Product>`. You're able to call it due to the presence of an *extension method*, which you'll see in more detail in chapter 10. You're not actually sorting the list "in place" anymore, just retrieving the contents

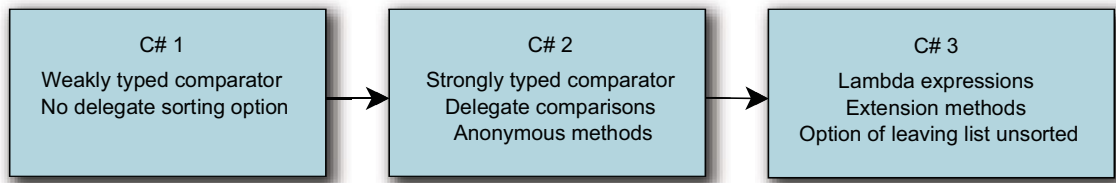


Figure 1.2 Features involved in making sorting easier in C# 2 and 3

of the list in a particular order. Sometimes you'll need to change the actual list; sometimes an ordering without any other side effects is better.

The important point is that this code is much more compact and readable (once you understand the syntax, of course). We wanted the list ordered by name, and that's exactly what the code says. It doesn't say to sort by comparing the name of one product with the name of another, like the C# 2 code did, or to sort by using an instance of another type that knows how to compare one product with another. It just says to order by name. This simplicity of expression is one of the key benefits of C# 3. When the individual pieces of data querying and manipulation are so simple, larger transformations can remain compact and readable in one piece of code. That, in turn, encourages a more data-centric way of looking at the world.

You've seen more of the power of C# 2 and 3 in this section, with a lot of (as yet) unexplained syntax, but even without understanding the details you can see the progress toward clearer, simpler code. Figure 1.2 shows that evolution.

That's it for sorting.² Let's do a different form of data manipulation now—querying.

1.2.2 Querying collections

Your next task is to find all the elements of the list that match a certain criterion—in particular, those with a price greater than \$10. The following listing shows how, in C# 1, you need to loop around, testing each element and printing it out when appropriate.

Listing 1.10 Looping, testing, printing out (C# 1)

```

ArrayList products = Product.GetSampleProducts();
foreach (Product product in products)
{
    if (product.Price > 10m)
    {
        Console.WriteLine(product);
    }
}
  
```

This code is *not* difficult to understand. But it's worth bearing in mind how intertwined the three tasks are—looping with `foreach`, testing the criterion with `if`, and

² C# 4 does provide one feature that can be relevant when sorting, called *generic variance*, but giving an example here would require too much explanation. You can find the details near the end of chapter 13.

then displaying the product with `Console.WriteLine`. The dependency is obvious because of the nesting.

The following listing demonstrates how C# 2 lets you flatten things out a bit.

Listing 1.11 Separating testing from printing (C# 2)

```
List<Product> products = Product.GetSampleProducts();  
  
Predicate<Product> test = delegate(Product p) { return p.Price > 10m; };  
List<Product> matches = products.FindAll(test);  
  
Action<Product> print = Console.WriteLine;  
matches.ForEach(print);
```

The `test` variable is initialized using the anonymous method feature you saw in the previous section. The `print` variable initialization uses another new C# 2 feature called *method group conversions* that makes it easier to create delegates from existing methods.

I'm not going to claim that this code is simpler than the C# 1 code, but it *is* a lot more powerful.³

In particular, the technique of separating the two concerns like this makes it *very* easy to change the condition you're testing for and the action you take on each of the matches independently. The delegate variables involved (`test` and `print`) could be passed into a method, and that same method could end up testing radically different conditions and taking radically different actions. Of course, you could put all the testing and printing into one statement, as shown in the following listing.

Listing 1.12 Separating testing from printing redux (C# 2)

```
List<Product> products = Product.GetSampleProducts();  
products.FindAll(delegate(Product p) { return p.Price > 10; })  
    .ForEach(Console.WriteLine);
```

In some ways, this version is better, but the `delegate(Product p)` is getting in the way, as are the braces. They're adding noise to the code, which hurts readability. I still prefer the C# 1 version in cases where I only ever want to use the same test and perform the same action. (It may sound obvious, but it's worth remembering that there's nothing stopping you from using the C# 1 code with a later compiler version. You wouldn't use a bulldozer to plant tulip bulbs, which is the kind of overkill used in the last listing.)

The next listing shows how C# 3 improves matters dramatically by removing a lot of the fluff surrounding the actual *logic* of the delegate.

³ In some ways, this is cheating. You could've defined appropriate delegates in C# 1 and called them within the loop. The `FindAll` and `ForEach` methods in .NET 2.0 just encourage you to consider separation of concerns.