

Covers C# 6 and 7



# C#

## IN DEPTH

FOURTH EDITION

Jon Skeet

FOREWORD BY ERIC LIPPETT

 MANNING

## Praise for the Third Edition

*“A must-have book that every .NET developer should read at least once.”*

—Dror Helper, Software Architect, Better Place

*“C# in Depth is the best source for learning C# language features.”*

—Andy Kirsch, Software Architect, Venga

*“Took my C# knowledge to the next level.”*

—Dustin Laine, Owner, Code Harvest

*“This book was quite an eye-opener to an interesting programming language that I have been unjustly ignoring until now.”*

—Ivan Todorović, Senior Software Developer  
AudatexGmbH, Switzerland

*“Easily the best C# reference I’ve found.”*

—Jon Parish, Software Engineer, Datasift

*“Highly recommend this book to C# developers who want to take their knowledge to pro status.”*

—D. Jay, Amazon reviewer

## Praise for the Second Edition

*“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*

*“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

*“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*

FOURTH EDITION

JON SKEET

FOREWORD BY ERIC LIPPERT



MANNING  
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit [www.manning.com](http://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 761  
Shelter Island, NY 11964  
Email: [orders@manning.com](mailto:orders@manning.com)

©2019 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 761  
Shelter Island, NY 11964

Development editor: Richard Wattenberger  
Technical development editor: Dennis Sellinger  
Reviewed editor: Ivan Martinović  
Production editor: Lori Weidert  
Copy editor: Sharon Wilkey  
Technical proofreader: Eric Lippert  
Typesetter and cover designer: Marija Tudor

ISBN 9781617294532

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – SP – 24 23 22 21 20 19

*This book is dedicated to equality, which is significantly harder to achieve in the real world than overriding `Equals()` and `GetHashCode()`.*



# contents

---

<i>foreword</i>	xvii
<i>preface</i>	xix
<i>acknowledgments</i>	xx
<i>about this book</i>	xxii
<i>about the author</i>	xxvi
<i>about the cover illustration</i>	xxvii

## PART 1 C# IN CONTEXT ..... 1

### 1 *Survival of the sharpest* 3

1.1	An evolving language	3		
	<i>A helpful type system at large and small scales</i>	4 ■ <i>Ever more concise code</i>	6 ■ <i>Simple data access with LINQ</i>	9
	<i>Asynchrony</i>	10 ■ <i>Balancing efficiency and complexity</i>	11	
	<i>Evolution at speed: Using minor versions</i>	12		
1.2	An evolving platform	13		
1.3	An evolving community	14		
1.4	An evolving book	15		
	<i>Mixed-level coverage</i>	16 ■ <i>Examples using Noda Time</i>	16	
	<i>Terminology choices</i>	17		



## PART 2 C# 2–5 ..... 19

### 2 C# 2 21

#### 2.1 Generics 22

*Introduction by example: Collections before generics* 22  
*Generics save the day* 25 ▪ *What can be generic?* 29  
*Type inference for type arguments to methods* 30 ▪ *Type constraints* 32 ▪ *The default and typeof operators* 34  
*Generic type initialization and state* 37

#### 2.2 Nullable value types 38

*Aim: Expressing an absence of information* 39 ▪ *CLR and framework support: The Nullable<T> struct* 40 ▪ *Language support* 43

#### 2.3 Simplified delegate creation 49

*Method group conversions* 50 ▪ *Anonymous methods* 50  
*Delegate compatibility* 52

#### 2.4 Iterators 53

*Introduction to iterators* 54 ▪ *Lazy execution* 55 ▪ *Evaluation of yield statements* 56 ▪ *The importance of being lazy* 57  
*Evaluation of finally blocks* 58 ▪ *The importance of finally handling* 61 ▪ *Implementation sketch* 62

#### 2.5 Minor features 66

*Partial types* 67 ▪ *Static classes* 69 ▪ *Separate getter/setter access for properties* 69 ▪ *Namespace aliases* 70  
*Pragma directives* 72 ▪ *Fixed-size buffers* 73  
*InternalsVisibleTo* 73

### 3 C# 3: LINQ and everything that comes with it 75

#### 3.1 Automatically implemented properties 76

#### 3.2 Implicit typing 77

*Typing terminology* 77 ▪ *Implicitly typed local variables (var)* 78 ▪ *Implicitly typed arrays* 79

#### 3.3 Object and collection initializers 81

*Introduction to object and collection initializers* 81  
*Object initializers* 83 ▪ *Collection initializers* 84  
*The benefits of single expressions for initialization* 86

#### 3.4 Anonymous types 86

*Syntax and basic behavior* 86 ▪ *The compiler-generated type* 89 ▪ *Limitations* 90

- 3.5 Lambda expressions 91
  - Lambda expression syntax* 92 ▪ *Capturing variables* 94
  - Expression trees* 101
- 3.6 Extension methods 103
  - Declaring an extension method* 103 ▪ *Invoking an extension method* 104 ▪ *Chaining method calls* 106
- 3.7 Query expressions 107
  - Query expressions translate from C# to C#* 108 ▪ *Range variables and transparent identifiers* 108 ▪ *Deciding when to use which syntax for LINQ* 110
- 3.8 The end result: LINQ 111

## 4 C# 4: Improving interoperability 113

- 4.1 Dynamic typing 114
  - Introduction to dynamic typing* 114 ▪ *Dynamic behavior beyond reflection* 119 ▪ *A brief look behind the scenes* 124
  - Limitations and surprises in dynamic typing* 127 ▪ *Usage suggestions* 131
- 4.2 Optional parameters and named arguments 133
  - Parameters with default values and arguments with names* 134
  - Determining the meaning of a method call* 135 ▪ *Impact on versioning* 137
- 4.3 COM interoperability improvements 138
  - Linking primary interop assemblies* 139 ▪ *Optional parameters in COM* 140 ▪ *Named indexers* 142
- 4.4 Generic variance 143
  - Simple examples of variance in action* 143 ▪ *Syntax for variance in interface and delegate declarations* 144
  - Restrictions on using variance* 145 ▪ *Generic variance in practice* 147

## 5 Writing asynchronous code 150

- 5.1 Introducing asynchronous functions 152
  - First encounters of the asynchronous kind* 152 ▪ *Breaking down the first example* 154
- 5.2 Thinking about asynchrony 155
  - Fundamentals of asynchronous execution* 155 ▪ *Synchronization contexts* 157 ▪ *Modeling asynchronous methods* 158

- 5.3 Async method declarations 160
  - Return types from async methods* 161 ▪ *Parameters in async methods* 162
- 5.4 Await expressions 162
  - The awaitable pattern* 163 ▪ *Restrictions on await expressions* 165
- 5.5 Wrapping of return values 166
- 5.6 Asynchronous method flow 168
  - What is awaited and when?* 168 ▪ *Evaluation of await expressions* 169 ▪ *The use of awaitable pattern members* 173
  - Exception unwrapping* 174 ▪ *Method completion* 176
- 5.7 Asynchronous anonymous functions 180
- 5.8 Custom task types in C# 7 182
  - The 99.9% case: ValueTask<TResult>* 182 ▪ *The 0.1% case: Building your own custom task type* 184
- 5.9 Async main methods in C# 7.1 186
- 5.10 Usage tips 187
  - Avoid context capture by using ConfigureAwait (where appropriate)* 187 ▪ *Enable parallelism by starting multiple independent tasks* 189 ▪ *Avoid mixing synchronous and asynchronous code* 190 ▪ *Allow cancellation wherever possible* 190 ▪ *Testing asynchrony* 191

## 6 Async implementation 193

- 6.1 Structure of the generated code 195
  - The stub method: Preparation and taking the first step* 198
  - Structure of the state machine* 199 ▪ *The MoveNext() method (high level)* 202 ▪ *The SetStateMachine method and the state machine boxing dance* 204
- 6.2 A simple MoveNext() implementation 205
  - A full concrete example* 205 ▪ *MoveNext() method general structure* 207 ▪ *Zooming into an await expression* 209
- 6.3 How control flow affects MoveNext() 210
  - Control flow between await expressions is simple* 211
  - Awaiting within a loop* 212 ▪ *Awaiting within a try/finally block* 213
- 6.4 Execution contexts and flow 216
- 6.5 Custom task types revisited 218

## 7 C# 5 bonus features 220

### 7.1 Capturing variables in foreach loops 220

### 7.2 Caller information attributes 222

*Basic behavior* 222 ▪ *Logging* 224 ▪ *Simplifying  
INotifyPropertyChanged implementations* 224 ▪ *Corner cases of  
caller information attributes* 226 ▪ *Using caller information  
attributes with old versions of .NET* 232

## PART 3 C# 6 ..... 233

## 8 Super-sleek properties and expression-bodied members 235

### 8.1 A brief history of properties 236

### 8.2 Upgrades to automatically implemented properties 238

*Read-only automatically implemented properties* 238

*Initializing automatically implemented properties* 239

*Automatically implemented properties in structs* 240

### 8.3 Expression-bodied members 242

*Even simpler read-only computed properties* 242 ▪ *Expression-  
bodied methods, indexers, and operators* 245 ▪ *Restrictions on  
expression-bodied members in C# 6* 247 ▪ *Guidelines for using  
expression-bodied members* 249

## 9 Stringy features 252

### 9.1 A recap on string formatting in .NET 253

*Simple string formatting* 253 ▪ *Custom formatting with format  
strings* 253 ▪ *Localization* 255

### 9.2 Introducing interpolated string literals 258

*Simple interpolation* 258 ▪ *Format strings in interpolated string  
literals* 259 ▪ *Interpolated verbatim string literals* 259

*Compiler handling of interpolated string literals (part 1)* 261

### 9.3 Localization using FormattableString 261

*Compiler handling of interpolated string literals (part 2)* 262

*Formatting a FormattableString in a specific culture* 263

*Other uses for FormattableString* 265 ▪ *Using FormattableString  
with older versions of .NET* 268

### 9.4 Uses, guidelines, and limitations 270

*Developers and machines, but maybe not end users* 270

*Hard limitations of interpolated string literals* 272 ▪ *When you  
can but really shouldn't* 273

- 9.5 Accessing identifiers with `nameof` 275
  - First examples of `nameof`* 275 ▪ *Common uses of `nameof`* 277
  - Tricks and traps when using `nameof`* 280

## 10 *A smörgåsbord of features for concise code* 284

- 10.1 Using static directives 284
  - Importing static members* 285 ▪ *Extension methods and using static* 288
- 10.2 Object and collection initializer enhancements 290
  - Indexers in object initializers* 291 ▪ *Using extension methods in collection initializers* 294 ▪ *Test code vs. production code* 298
- 10.3 The null conditional operator 299
  - Simple and safe property dereferencing* 299 ▪ *The null conditional operator in more detail* 300 ▪ *Handling Boolean comparisons* 301 ▪ *Indexers and the null conditional operator* 302 ▪ *Working effectively with the null conditional operator* 303 ▪ *Limitations of the null conditional operator* 305
- 10.4 Exception filters 305
  - Syntax and semantics of exception filters* 306 ▪ *Retrying operations* 311 ▪ *Logging as a side effect* 312 ▪ *Individual, case-specific exception filters* 313 ▪ *Why not just throw?* 314

## PART 4 C# 7 AND BEYOND ..... 317

## 11 *Composition using tuples* 319

- 11.1 Introduction to tuples 320
- 11.2 Tuple literals and tuple types 321
  - Syntax* 321 ▪ *Inferred element names for tuple literals (C# 7.1)* 323 ▪ *Tuples as bags of variables* 324
- 11.3 Tuple types and conversions 329
  - Types of tuple literals* 329 ▪ *Conversions from tuple literals to tuple types* 330 ▪ *Conversions between tuple types* 334 ▪ *Uses of conversions* 336 ▪ *Element name checking in inheritance* 336
  - Equality and inequality operators (C# 7.3)* 337
- 11.4 Tuples in the CLR 338
  - Introducing `System.ValueTuple<...>`* 338 ▪ *Element name handling* 339 ▪ *Tuple conversion implementations* 341
  - String representations of tuples* 341 ▪ *Regular equality and ordering comparisons* 342 ▪ *Structural equality and ordering*

- comparisons* 343 ▪ *Womples and large tuples* 345 ▪ *The nongeneric ValueTuple struct* 346 ▪ *Extension methods* 346
- 11.5 Alternatives to tuples 346
  - System.Tuple<...>* 347 ▪ *Anonymous types* 347
  - Named types* 348
- 11.6 Uses and recommendations 348
  - Nonpublic APIs and easily changed code* 348 ▪ *Local variables* 349 ▪ *Fields* 350 ▪ *Tuples and dynamic don't play together nicely* 351

## 12 Deconstruction and pattern matching 353

- 12.1 Deconstruction of tuples 354
  - Deconstruction to new variables* 355 ▪ *Deconstruction assignments to existing variables and properties* 357
  - Details of tuple literal deconstruction* 361
- 12.2 Deconstruction of nontuple types 361
  - Instance deconstruction methods* 362 ▪ *Extension deconstruction methods and overloading* 363 ▪ *Compiler handling of Deconstruct calls* 364
- 12.3 Introduction to pattern matching 365
- 12.4 Patterns available in C# 7.0 367
  - Constant patterns* 367 ▪ *Type patterns* 368 ▪ *The var pattern* 371
- 12.5 Using patterns with the is operator 372
- 12.6 Using patterns with switch statements 374
  - Guard clauses* 375 ▪ *Pattern variable scope for case labels* 376 ▪ *Evaluation order of pattern-based switch statements* 377
- 12.7 Thoughts on usage 379
  - Spotting deconstruction opportunities* 379 ▪ *Spotting pattern matching opportunities* 380

## 13 Improving efficiency with more pass by reference 381

- 13.1 Recap: What do you know about ref? 382
- 13.2 Ref locals and ref returns 385
  - Ref locals* 385 ▪ *Ref returns* 390 ▪ *The conditional ?: operator and ref values (C# 7.2)* 392 ▪ *Ref readonly (C# 7.2)* 393
- 13.3 in parameters (C# 7.2) 395

*Compatibility considerations* 396 ▪ *The surprising mutability of in parameters: External changes* 397 ▪ *Overloading with in parameters* 398 ▪ *Guidance for in parameters* 399

#### 13.4 Declaring structs as readonly (C# 7.2) 401

*Background: Implicit copying with read-only variables* 401  
*The readonly modifier for structs* 403 ▪ *XML serialization is implicitly read-write* 404

#### 13.5 Extension methods with ref or in parameters (C# 7.2) 405

*Using ref/in parameters in extension methods to avoid copying* 405  
*Restrictions on ref and in extension methods* 407

#### 13.6 Ref-like structs (C# 7.2) 408

*Rules for ref-like structs* 409 ▪ *Span<T> and stackalloc* 410  
*IL representation of ref-like structs* 414

## 14 Concise code in C# 7 415

#### 14.1 Local methods 415

*Variable access within local methods* 417 ▪ *Local method implementations* 420 ▪ *Usage guidelines* 425

#### 14.2 Out variables 427

*Inline variable declarations for out parameters* 427 ▪ *Restrictions lifted in C# 7.3 for out variables and pattern variables* 428

#### 14.3 Improvements to numeric literals 429

*Binary integer literals* 429 ▪ *Underscore separators* 430

#### 14.4 Throw expressions 431

#### 14.5 Default literals (C# 7.1) 432

#### 14.6 Nontrailing named arguments (C# 7.2) 433

#### 14.7 Private protected access (C# 7.2) 435

#### 14.8 Minor improvements in C# 7.3 435

*Generic type constraints* 435 ▪ *Overload resolution improvements* 436 ▪ *Attributes for fields backing automatically implemented properties* 437

## 15 C# 8 and beyond 439

#### 15.1 Nullable reference types 440

*What problem do nullable reference types solve?* 440 ▪ *Changing the meaning when using reference types* 441 ▪ *Enter nullable reference types* 442 ▪ *Nullable reference types at compile time and*

	<i>execution time</i>	443	▪	<i>The damn it or bang operator</i>	445
	<i>Experiences of nullable reference type migration</i>	447			
	<i>Future improvements</i>	449			
15.2	Switch expressions	453			
15.3	Recursive pattern matching	455			
	<i>Matching properties in patterns</i>	455	▪	<i>Deconstruction patterns</i>	456
			▪	<i>Omitting types from patterns</i>	457
15.4	Indexes and ranges	458			
	<i>Index and Range types and literals</i>	458	▪	<i>Applying indexes and ranges</i>	459
15.5	More async integration	461			
	<i>Asynchronous resource disposal with using await</i>	461			
	<i>Asynchronous iteration with foreach await</i>	462	▪	<i>Asynchronous iterators</i>	465
15.6	Features not yet in preview	466			
	<i>Default interface methods</i>	466	▪	<i>Record types</i>	468
	<i>Even more features in brief</i>	469			
15.7	Getting involved	470			
appendix	<i>Language features by version</i>	473			
	<i>index</i>	479			





## *foreword*

---

Ten years is a long stretch of time for a human, and it's an absolute eternity for a technical book aimed at professional programmers. It was with some astonishment, then, that I realized 10 years have passed since Microsoft shipped C# 3.0 with Visual Studio 2008 and since I read the drafts of the first edition of this book. It has also been 10 years since Jon joined Stack Overflow and quickly became the user with the highest reputation.

C# was already a large, complex language in 2008, and the design and implementation teams haven't been idle for the last decade. I'm thrilled with how C# has been innovative in meeting the needs of many different developer constituencies, from video games to websites to low-level, highly robust system components. C# takes the best from academic research and marries it to practical techniques for solving real problems. It's not dogmatic; the C# designers don't ask "What's the most object-oriented way to design this feature?" or "What's the most functional way to design this feature?" but rather "What's the most pragmatic, safe, and effective way to design this feature?" Jon gets all of that. He doesn't just explain how the language works; he explains how the whole thing holds together as a unified design and also points out when it doesn't.

I said in my foreword to the first edition that Jon is enthusiastic, knowledgeable, talented, curious, analytical, and a great teacher, and all of that is still true. Let me add to that list by noting his perseverance and dedication. Writing a book is a huge job, particularly when you do it in your spare time. Going back and revising that book to keep it fresh and current is just as much work, and this is the third time Jon has done that with this book. A lesser author would be content to tweak it here and there or add

a chapter about new materials; this is more like a large-scale refactoring. The results speak for themselves.

More than ever, I can't wait to find out what great things the next generation of programmers will do with C# as it continues to evolve and grow. I hope you enjoy this book as much as I have over the years, and thanks for choosing to compose your programs in C#.

ERIC LIPPERT  
SOFTWARE ENGINEER  
FACEBOOK

## *preface*

---

Welcome to the fourth edition of *C# in Depth*. When I wrote the first edition, I had little idea I'd be writing a fourth edition of the same title 10 years later. Now, it wouldn't surprise me to find myself writing another edition in 10 years. Since the first edition, the designers of the C# language have repeatedly proved that they're dedicated to evolving the language for as long as the industry is interested in it.

This is important, because the industry has changed a lot in the last 10 years. As a reminder, both the mobile ecosystem (as we know it today) and cloud computing were still in their infancy in 2008. Amazon EC2 was launched in 2006, and Google AppEngine was launched in 2008. Xamarin was launched by the Mono team in 2011. Docker didn't show up until 2013.

For many .NET developers, the really big change in our part of the computing world over the last few years has been .NET Core. It's a cross-platform, open source version of the framework that is explicitly designed for compatibility with other frameworks (via .NET Standard). Its existence is enough to raise eyebrows; that it is Microsoft's primary area of investment in .NET is even more surprising.

Through all of this, C# is still the primary language when targeting anything like .NET, whether that's .NET, .NET Core, Xamarin, or Unity. F# is a healthy and friendly competitor, but it doesn't have the industry mindshare of C#.

I've personally been developing in C# since around 2002, either professionally or as an enthusiastic amateur. As the years have gone by, I've been sucked ever deeper into the details of the language. I enjoy those details for their own sake but, more importantly, for the sake of ever-increasing productivity when writing code in C#. I hope that some of that enjoyment has seeped into this book and will encourage you further in your travels with C#.

## *acknowledgments*

---

It takes a lot of work and energy to create a book. Some of that is obvious; after all, pages don't just write themselves. That's just the tip of the iceberg, though. If you received the first version of the content I wrote with no editing, no review, no professional typesetting, and so on, I suspect you'd be pretty disappointed.

As with previous editions, it's been a pleasure working with the team at Manning. Richard Wattenberger has provided guidance and suggestions with just the right combination of insistence and understanding, thereby shaping the content through multiple iterations. (In particular, working out the best approach to use for C# 2–4 proved surprisingly challenging.) I would also like to thank Mike Stephens and Marjan Bace for supporting this edition from the start.

Beyond the structure of the book, the review process is crucial to keeping the content accurate and clear. Ivan Martinovic organized the peer reviewing process and obtained great feedback from Ajay Bhosale, Andrei Rînea, Andy Kirsch, Brian Rasmussen, Chris Heneghan, Christos Paisios, Dmytro Lypai, Ernesto Cardenas, Gary Hubbard, Jassel Holguin Calderon, Jeremy Lange, John Meyer, Jose Luis Perez Vila, Karl Metivier, Meredith Godar, Michal Paszkiewicz, Mikkel Arentoft, Nelson Ferrari, Prajwal Khanal, Rami Abdelwahed, and Willem van Ketwicha. I'm indebted to Dennis Sellinger for his technical editing and to Eric Lippert for technical proofreading. I want to highlight Eric's contributions to every edition of this book, which have always gone well beyond technical corrections. His insight, experience, and humor have been significant and unexpected bonuses throughout the whole process.

Content is one thing; good-looking content is another. Lori Weidert managed the complex production process with dedication and understanding. Sharon Wilkey performed copyediting with skill and the utmost patience. The typesetting and cover

design were done by Marija Tudor, and I can't express what a joy it is to see the first typeset pages; it's much like the first (successful) dress rehearsal of a play you've been working on for months.

Beyond the people who've contributed directly to the book, I naturally need to thank my family for continuing to put up with me over the last few years. I love my family. They rock, and I'm grateful.

Finally, none of this would matter if no one wanted to read the book. Thank you for your interest, and I hope your investment of time into this book pays off.

# about this book

---

## **Who should read this book**

This book is about the *language* of C#. That often means going into some details of the runtime responsible for executing your code and the libraries that support your application, but the focus is firmly on the language itself.

The goal of the book is to make you as comfortable as possible with C# so you never need to feel you're fighting against it. I want to help you feel you are *fluent* in C#, with the associated connotations of working in a fluid and flowing way. Think of C# as a river in which you're paddling a kayak. The better you know the river, the faster you'll be able to travel with its flow. Occasionally, you'll want to paddle upstream for some reason; even then, knowing how the river moves will make it easier to reach your target without capsizing.

If you're an existing C# programmer who wants to know more about the language, this book is for you! You don't need to be an expert to read this book, but I assume you know the basics of C# 1. I explain all the terminology I use that was introduced after C# 1 and some older terms that are often misunderstood (such as parameters and arguments), but I assume you know what a class is, what an object is, and so on.

If you are an expert already, you may still find the book useful because it provides different ways of thinking about concepts that are already familiar to you. You may also discover areas of the language you were unaware of; I know that's been my experience in writing the book.

If you're completely new to C#, this book may not be useful to you *yet*. There are a lot of introductory books and online tutorials on C#. Once you have a grip on the basics, I hope you'll return here and dive deeper.

### ***How this book is organized: A roadmap***

This book comprises 15 chapters divided into 4 parts. Part 1 provides a brief history of the language.

- Chapter 1 gives an overview of how C# has changed over the years and how it is still changing. It puts C# into a broader context of platforms and communities and gives a little more detail about how I present material in the rest of the book.

Part 2 describes C# versions 2 through 5. This is effectively a rewritten and condensed form of the third edition of this book.

- Chapter 2 demonstrates the wide variety of features introduced in C# 2, including generics, nullable value types, anonymous methods, and iterators.
- Chapter 3 explains how the features of C# 3 come together to form LINQ. The most prominent features in this chapter are lambda expressions, anonymous types, object initializers, and query expressions.
- Chapter 4 describes the features of C# 4. The largest change within C# 4 was the introduction of dynamic typing, but there are other changes around optional parameters, named arguments, generic variance, and reducing friction when working with COM.
- Chapter 5 begins the coverage of C# 5's primary feature: `async/await`. This chapter describes how you'll use `async/await` but has relatively little detail about how it works behind the scenes. Enhancements to asynchrony introduced in later versions of C# are described here as well, including custom task types and `async` main methods.
- Chapter 6 completes the `async/await` coverage by going deep into the details of how the compiler handles asynchronous methods by creating state machines.
- Chapter 7 is a short discussion of the few features introduced in C# 5 besides `async/await`. After the all the details provided in chapter 6, you can consider it a palette cleanser before moving on to the next part of the book.

Part 3 describes C# 6 in detail.

- Chapter 8 shows expression-bodied members, which allow you to remove some of the tedious syntax when declaring very simple properties and methods. Improvements to automatically implemented properties are described here, too. It's all about streamlining your source code.
- Chapter 9 describes the string-related features of C# 6: interpolated string literals and the `nameof` operator. Although both features are just new ways of producing strings, they are among the most handy aspects of C# 6.
- Chapter 10 introduces the remaining features of C# 6. These have no particularly common theme other than helping you write concise source code. Of the



features introduced here, the null conditional operator is probably the most useful; it's a clean way of short-circuiting expressions that might involve null values, thereby avoiding the dreaded `NullReferenceException`.

Part 4 addresses C# 7 (all the way up to C# 7.3) and completes the book by peering a short distance into the future.

- Chapter 11 demonstrates the integration of tuples into the language and describes the `ValueTuple` family of types that is used for the implementation.
- Chapter 12 introduces deconstruction and pattern matching. These are both concise ways of looking at an existing value in a different way. In particular, pattern matching in switch statements can simplify how you handle different types of values in situations where inheritance doesn't quite fit.
- Chapter 13 focuses on pass by reference and related features. Although ref parameters have been present in C# since the very first version, C# 7 introduces a raft of new features such as ref returns and ref locals. These are primarily aimed at improving efficiency by reducing copying.
- Chapter 14 completes the C# 7 coverage with another set of small features that all contribute to streamlining your code. Of these, my personal favorites are local methods, out variables, and the default literal, but there are other little gems to discover, too.
- Chapter 15 looks at the future of C#. Working with the C# 8 preview available at the time of this writing, I delve into nullable reference types, switch expressions, and pattern matching enhancements as well as ranges and further integration of asynchrony into core language features. This entire chapter is speculative, but I hope it will spark your curiosity.

Finally, the appendix provides a handy reference for which features were introduced in which version of C# and whether they have runtime or framework requirements that restrict the contexts in which you can use them.

My expectation is that this book will be read in a linear fashion (at least the first time). Later chapters build on earlier ones, and you may have a hard time if you try to read them out of order. After you've read the book once, however, it makes perfect sense to use it as a reference. You might go back to a topic when you need a reminder of some syntax or if you find yourself caring more about a specific detail than you did on your first reading.

### **About the code**

This book contains many examples of source code in numbered listings and in line with normal text. In both cases, source code is formatted in a fixed-width font like this to separate it from ordinary text. Sometimes it appears **in bold** to highlight code that has changed from previous steps in the chapter, such as when a new feature adds to an existing line of code.

In many cases, the original source code has been reformatted; I've added line breaks and reworked indentation to accommodate the available page space in the book. In rare cases, listings include line-continuation markers (➡). In addition, comments in the source code have often been removed from the listings when the code is described in the text. Code annotations accompany many of the listings and highlight important concepts.

Source code for the examples in this book is available for download from the publisher's website at [www.manning.com/books/c-sharp-in-depth-fourth-edition](http://www.manning.com/books/c-sharp-in-depth-fourth-edition). You'll need the .NET Core SDK (version 2.1.300 or higher) installed to build the examples. A few examples require the Windows desktop .NET framework (where Windows Forms or COM is involved), but most are portable via .NET Core. Although I used Visual Studio 2017 (Community Edition) to develop the examples, they should be fine under Visual Studio Code as well.

### **Book forum**

Purchase of *C# in Depth*, Fourth 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 from other users. To access the forum, go to <https://forums.manning.com/forums/c-sharp-in-depth-fourth-edition>. You can also learn more about Manning's forums and the rules of conduct at <https://forums.manning.com/forums/about>.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

### **Other online resources**

There are many, many resources for C# online. The ones I find most useful are listed below, but you'll find a lot more by searching, too.

- Microsoft .NET documentation: <https://docs.microsoft.com/dotnet>
- The .NET API documentation: <https://docs.microsoft.com/dotnet/api>
- The C# language design repository: <https://github.com/dotnet/csharplang>
- The Roslyn repository: <https://github.com/dotnet/roslyn>
- The C# ECMA standard:  
[www.ecma-international.org/publications/standards/Ecma-334.htm](http://www.ecma-international.org/publications/standards/Ecma-334.htm)
- Stack Overflow: <https://stackoverflow.com>

## *about the author*

---

My name is Jon Skeet. I'm a staff software engineer at Google, and I work from the London office. Currently, my role is to provide .NET client libraries for Google Cloud Platform, which neatly combines my enthusiasm for working at Google with my love of C#. I'm the convener of the ECMA technical group responsible for standardizing C#, and I represent Google within the .NET Foundation.

I'm probably best known for my contributions on Stack Overflow, which is a question-and-answer site for developers. I also enjoy speaking at conferences and user groups and blogging. The common factor here is interacting with other developers; it's the way I learn best.

Slightly more unusually, I'm a date and time hobbyist. This is mostly expressed through my work on Noda Time, which is the date and time library for .NET that you'll see used in several examples in this book. Even without the hands-on coding aspect, time is a fascinating topic with an abundance of trivia. Find me at a conference and I'll bore you for as long as you like about time zones and calendar systems.

My editors would like you to know most of these things to prove that I'm qualified to write this book, but please don't mistake them for a claim of infallibility. Humility is a vital part of being an effective software engineer, and I screw up just like everyone else does. Compilers don't tend to view appeals to authority in a favorable light.

In the book, I've tried to make it clear where I'm expressing what I believe to be objective facts about the C# language and where I'm expressing my opinion. Due to diligent technical reviewers, I hope there are relatively few mistakes on the objective side, but experience from previous editions suggests that some errors will have crept through. When it comes to opinions, mine may be wildly different from yours, and that's fine. Take what you find useful, and feel free to ignore the rest.

## *about the cover illustration*

---

The caption for the illustration on the cover of *C# in Depth*, Fourth 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, he transferred the funds the next day, and he remains grateful and impressed by this unknown person’s trust. 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*

## *C# in context*

When I was studying computer science at university, a fellow student corrected the lecturer about a detail he'd written on the blackboard. The lecturer looked mildly exasperated and answered, "Yes, I know. I was simplifying. I'm obscuring the truth here to demonstrate a bigger truth." Although I hope I'm not obscuring much in part 1, it's definitely about the bigger truth.

Most of this book looks at C# close up, occasionally putting it under a microscope to see the finest details. Before we start doing that, chapter 1 pulls back the lens to see the broader sweep of the history of C# and how C# fits into the wider context of computing.

You'll see some code as an appetizer before I serve the main course of the rest of the book, but the details don't matter at this stage. This part is more about the ideas and themes of C#'s development to get you in the best frame of mind to appreciate how those ideas are implemented.

Let's go!



# *Survival of the sharpest*

---



## ***This chapter covers***

- How C#'s rapid evolution has made developers more productive
- Selecting minor versions of C# to use the latest features
- Being able to run C# in more environments
- Benefitting from an open and engaged community
- The book's focus on old and new C# versions

Choosing the most interesting aspects of C# to introduce here was difficult. Some are fascinating but are rarely used. Others are incredibly important but are now commonplace to C# developers. Features such as `async/await` are great in many ways but are hard to describe briefly. Without further ado, let's look at how far C# has come over time.

## **1.1 *An evolving language***

In previous editions of this book, I provided a single example that showed the evolution of the language over the versions covered by that edition. That's no longer feasible in a way that would be interesting to read. Although a large application



may use almost all of the new features, any single piece of code that's suitable for the printed page would use only a subset of them.

Instead, in this section I choose what I consider to be the most important themes of C# evolution and give brief examples of improvements. This is far from an exhaustive list of features. It's also not intended to teach you the features; instead, it's a reminder of how far features you already know about have improved the language and a tease for features you may not have seen yet.

If you think some of these features imitate other languages you're familiar with, you're almost certainly right. The C# team does not hesitate to take great ideas from other languages and reshape them to feel at home within C#. This is a great thing! F# is particularly worth mentioning as a source of inspiration for many C# features.

**NOTE** It's possible that F#'s greatest impact isn't what it enables for F# developers but its influence on C#. This isn't to underplay the value of F# as a language in its own right or to suggest that it shouldn't be used directly. But currently, the C# community is significantly larger than the F# community, and the C# community owes a debt of gratitude to F# for inspiring the C# team.

Let's start with one of the most important aspects of C#: its type system.

### 1.1.1 *A helpful type system at large and small scales*

C# has been a statically typed language from the start: your code specifies the types of variables, parameters, values returned from methods, and so on. The more precisely you can specify the shape of the data your code accepts and returns, the more the compiler can help you avoid mistakes.

That's particularly true as the application you're building grows. If you can see all the code for your whole program on one screen (or at least hold it all in your head at one time), a statically typed language doesn't have much benefit. As the scale increases, it becomes increasingly important that your code concisely and effectively communicates what it does. You can do that through documentation, but static typing lets you communicate in a machine-readable way.

As C# has evolved, its type system has allowed more fine-grained descriptions. The most obvious example of this is *generics*. In C# 1, you might have had code like this:

```
public class Bookshelf
{
    public IEnumerable Books { get { ... } }
}
```

What type is each item in the Books sequence? The type system doesn't tell you. With generics in C# 2, you can communicate more effectively:

```
public class Bookshelf
{
    public IEnumerable<Book> Books { get { ... } }
}
```

C# 2 also brought *nullable value types*, thereby allowing the absence of information to be expressed effectively without resorting to magic values such as `-1` for a collection index or `DateTime.MinValue` for a date.

C# 7 gave us the ability to tell the compiler that a user-defined struct should be immutable using `readonly` struct declarations. The primary goal for this feature may have been to improve the efficiency of the code generated by the compiler, but it has additional benefits for communicating intent.

The plans for C# 8 include *nullable reference types*, which will allow even more communication. Up to this point, nothing in the language lets you express whether a reference (either as a return value, a parameter, or just a local variable) might be null. This leads to error-prone code if you're not careful and boilerplate validation code if you are careful, neither of which is ideal. C# 8 will expect that anything not explicitly nullable is intended not to be nullable. For example, consider a method declaration like this:

```
string Method(string x, string? y)
```

The parameter types indicate that the argument corresponding to `x` shouldn't be null but that the argument corresponding to `y` may be null. The return type indicates that the method won't return null.

Other changes to the type system in C# are aimed at a smaller scale and focus on how one method might be implemented rather than how different components in a large system relate to each other. C# 3 introduced *anonymous types* and *implicitly typed local variables* (`var`). These help address the downside of some statically typed languages: verbosity. If you need a particular data shape within a single method but nowhere else, creating a whole extra type just for the sake of that method is overkill. Anonymous types allow that data shape to be expressed concisely without losing the benefits of static typing:

```
var book = new { Title = "Lost in the Snow", Author = "Holly Webb" };
string title = book.Title;
string author = book.Author;
```

Name and type are still checked by the compiler

Anonymous types are primarily used within LINQ queries, but the principle of creating a type just for a single method doesn't depend on LINQ.

Similarly, it seems redundant to explicitly specify the type of a variable that is initialized in the same statement by calling the constructor of that type. I know which of the following declarations I find cleaner:

```
Dictionary<string, string> map1 = new Dictionary<string, string>();
```

```
var map2 = new Dictionary<string, string>();
```

Implicit typing

Explicit typing

Although implicit typing is necessary when working with anonymous types, I've found it increasingly useful when working with regular types, too. It's important to distinguish

between *implicit* typing and *dynamic* typing. The preceding `map2` variable is still statically typed, but you didn't have to write the type explicitly.

Anonymous types help only within a single block of code; for example, you can't use them as method parameters or return types. C# 7 introduced *tuples*: value types that effectively act to collect variables together. The framework support for these tuples is relatively simple, but additional language support allows the elements of tuples to be named. For example, instead of the preceding anonymous type, you could use the following:

```
var book = (title: "Lost in the Snow", author: "Holly Webb");
Console.WriteLine(book.title);
```

Tuples can replace anonymous types in some cases but certainly not all. One of their benefits is that they *can* be used as method parameters and return types. At the moment, I advise that these be kept within the internal API of a program rather than exposed publicly, because tuples represent a simple composition of values rather than encapsulating them. That's why I still regard them as contributing to simpler code at the implementation level rather than improving overall program design.

I should mention a feature that *might* come in C# 8: *record types*. I think of these as named anonymous types to some extent, at least in their simplest form. They'd provide the benefits of anonymous types in terms of removing boilerplate code but then allow those types to gain extra behavior just as regular classes do. Watch this space!

### 1.1.2 *Ever more concise code*

One of the recurring themes within new features of C# has been the ability to let you express your ideas in ways that are increasingly concise. The type system is part of this, as you've seen with anonymous types, but many other features also contribute to this. There are lots of words you might hear for this, especially in terms of what can be removed with the new features in place. C#'s features allow you to reduce *ceremony*, remove *boilerplate* code, and avoid *cruft*. These are just different ways of talking about the same effect. It's not that any of the now-redundant code was wrong; it was just distracting and unnecessary. Let's look at a few ways that C# has evolved in this respect.

#### CONSTRUCTION AND INITIALIZATION

First, we'll consider how you create and initialize objects. Delegates have probably evolved the most and in multiple stages. In C# 1, you had to write a separate method for the delegate to refer to and then create the delegate itself in a long-winded way. For example, here's what you'd write to subscribe a new event handler to a button's Click event in C# 1:

```
button.Click += new EventHandler(HandleButtonClick);    ← C# 1
```

C# 2 introduced *method group conversions* and *anonymous methods*. If you wanted to keep the `HandleButtonClick` method, method group conversions would allow you to change the preceding code to the following:

```
button.Click += HandleButtonClick;                    ← C# 2
```

If your click handler is simple, you might not want to bother with a separate method at all and instead use an anonymous method:

```
button.Click += delegate { MessageBox.Show("Clicked!"); };    ← C# 2
```

Anonymous methods have the additional benefit of acting as *closures*: they can use local variables in the context within which they're created. They're not used often in modern C# code, however, because C# 3 provided us with *lambda expressions*, which have almost all the benefits of anonymous methods but shorter syntax:

```
button.Click += (sender, args) => MessageBox.Show("Clicked!");    ← C# 3
```

**NOTE** In this case, the lambda expression is longer than the anonymous method because the anonymous method uses the one feature that lambda expressions don't have: the ability to ignore parameters by not providing a parameter list.

I used event handlers as an example for delegates because that was their main use in C# 1. In later versions of C#, delegates are used in more varied situations, particularly in LINQ.

LINQ also brought other benefits for initialization in the form of *object initializers* and *collection initializers*. These allow you to specify a set of properties to set on a new object or items to add to a new collection within a single expression. It's simpler to show than describe, and I'll borrow an example from chapter 3. Consider code that you might previously have written like this:

```
var customer = new Customer();
customer.Name = "Jon";
customer.Address = "UK";
var item1 = new OrderItem();
item1.ItemId = "abcd123";
item1.Quantity = 1;
var item2 = new OrderItem();
item2.ItemId = "fghi456";
item2.Quantity = 2;
var order = new Order();
order.OrderId = "xyz";
order.Customer = customer;
order.Items.Add(item1);
order.Items.Add(item2);
```

The object and collection initializers introduced in C# 3 make this so much clearer:

```
var order = new Order
{
    OrderId = "xyz",
    Customer = new Customer { Name = "Jon", Address = "UK" },
    Items =
    {
        new OrderItem { ItemId = "abcd123", Quantity = 1 },
        new OrderItem { ItemId = "fghi456", Quantity = 2 }
    }
};
```

I don't suggest reading either of these examples in detail; what's important is the simplicity of the second form over the first.

#### METHOD AND PROPERTY DECLARATIONS

One of the most obvious examples of simplification is through *automatically implemented properties*. These were first introduced in C# 3 but have been further improved in later versions. Consider a property that would've been implemented in C# 1 like this:

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

Automatically implemented properties allow this to be written as a single line:

```
public string Name { get; set; }
```

Additionally, C# 6 introduced *expression-bodied members* that remove more ceremony. Suppose you're writing a class that wraps an existing collection of strings, and you want to effectively delegate the `Count` and `GetEnumerator()` members of your class to that collection. Prior to C# 6, you would've had to write something like this:

```
public int Count { get { return list.Count; } }

public IEnumerator<string> GetEnumerator()
{
    return list.GetEnumerator();
}
```

This is a strong example of ceremony: a lot of syntax that the language used to require with little benefit. In C# 6, this is significantly cleaner. The `=>` syntax (already used by lambda expressions) is used to indicate an expression-bodied member:

```
public int Count => list.Count;

public IEnumerator<string> GetEnumerator() => list.GetEnumerator();
```

Although the value of using expression-bodied members is a personal and subjective matter, I've been surprised by just how much difference they've made to the readability of my code. I love them! Another feature I hadn't expected to use as much as I now do is string interpolation, which is one of the string-related improvements in C#.

#### STRING HANDLING

String handling in C# has had three significant improvements:

- C# 5 introduced *caller information attributes*, including the ability for the compiler to automatically populate method and filenames as parameter values. This is great for diagnostic purposes, whether in permanent logging or more temporary testing.

- C# 6 introduced the `nameof` operator, which allows names of variables, types, methods, and other members to be represented in a refactoring-friendly form.
- C# 6 also introduced *interpolated string literals*. This isn't a new concept, but it makes constructing a string with dynamic values much simpler.

For the sake of brevity, I'll demonstrate just the last point. It's reasonably common to want to construct a string with variables, properties, the result of method calls, and so forth. This might be for logging purposes, user-oriented error messages (if localization isn't required), exception messages, and so forth.

Here's an example from my Noda Time project. Users can try to find a calendar system by its ID, and the code throws a `KeyNotFoundException` if that ID doesn't exist. Prior to C# 6, the code might have looked like this:

```
throw new KeyNotFoundException(
    "No calendar system for ID " + id + " exists");
```

Using explicit string formatting, it looks like this:

```
throw new KeyNotFoundException(
    string.Format("No calendar system for ID {0} exists", id);
```

**NOTE** See section 1.4.2 for information about Noda Time. You don't need to know about it to understand this example.

In C# 6, the code becomes just a little simpler with an interpolated string literal to include the value of `id` in the string directly:

```
throw new KeyNotFoundException($"No calendar system for ID {id} exists");
```

This doesn't look like a big deal, but I'd hate to have to work without string interpolation now.

These are just the most prominent features that help improve the signal-to-noise ratio of your code. I could've shown using `static` directives and the null conditional operator in C# 6 as well as pattern matching, deconstruction, and out variables in C# 7. Rather than expand this chapter to mention every feature in every version, let's move on to a feature that's more revolutionary than evolutionary: LINQ.

### 1.1.3 Simple data access with LINQ

If you ask C# developers what they love about C#, they'll likely mention LINQ. You've already seen some of the features that build up to LINQ, but the most radical is query expressions. Consider this code:

```
var offers =
    from product in db.Products
    where product.SalePrice <= product.Price / 2
    orderby product.SalePrice
    select new {
        product.Id, product.Description,
        product.SalePrice, product.Price
    };
```

That doesn't look anything like old-school C#. Imagine traveling back to 2007 to show that code to a developer using C# 2 and then explaining that this has compile-time checking and IntelliSense support and that it results in an efficient database query. Oh, and that you can use the same syntax for regular collections as well.

Support for querying out-of-process data is provided via *expression trees*. These represent code as data, and a LINQ provider can analyze the code to convert it into SQL or other query languages. Although this is extremely cool, I rarely use it myself, because I don't work with SQL databases often. I do work with in-memory collections, though, and I use LINQ all the time, whether through query expressions or method calls with lambda expressions.

LINQ didn't just give C# developers new tools; it encouraged us to think about data transformations in a new way based on functional programming. This affects more than data access. LINQ provided the initial impetus to take on more functional ideas, but many C# developers have embraced those ideas and taken them further.

C# 4 made a radical change in terms of dynamic typing, but I don't think that affected as many developers as LINQ. Then C# 5 came along and changed the game again, this time with respect to asynchrony.

### 1.1.4 **Asynchrony**

Asynchrony has been difficult in mainstream languages for a long time. More niche languages have been created with asynchrony in mind from the start, and some functional languages have made it relatively easy as just one of the things they handle neatly. But C# 5 brought a new level of clarity to programming asynchrony in a mainstream language with a feature usually referred to as *async/await*. The feature consists of two complementary parts around async methods:

- Async methods produce a result representing an asynchronous operation with no effort on the part of the developer. This result type is usually `Task` or `Task<T>`.
- Async methods use `await` expressions to consume asynchronous operations. If the method tries to await an operation that hasn't completed yet, the method pauses asynchronously until the operation completes and then continues.

**NOTE** More properly, I could call these asynchronous *functions*, because anonymous methods and lambda expressions can be asynchronous, too.

Exactly what's meant by *asynchronous operation* and *pausing asynchronously* is where things become tricky, and I won't attempt to explain this now. But the upshot is that you can write code that's asynchronous but looks mostly like the synchronous code you're more familiar with. It even allows for concurrency in a natural way. As an example, consider this asynchronous method that might be called from a Windows Forms event handler:

```
private async Task UpdateStatus()
{
    Task<Weather> weatherTask = GetWeatherAsync();
    Task<EmailStatus> emailTask = GetEmailStatusAsync();
```

**Starts two operations  
concurrently**

<pre> Weather weather = await weatherTask; EmailStatus email = await emailTask;  weatherLabel.Text = weather.Description; inboxLabel.Text = email.InboxCount.ToString(); } </pre>	<div style="border-left: 1px solid black; padding-left: 10px;"> <b>Asynchronously waits for them to complete</b> </div> <div style="border-left: 1px solid black; padding-left: 10px; margin-top: 20px;"> <b>Updates the userinterface</b> </div>
---	---

In addition to starting two operations concurrently and then awaiting their results, this demonstrates how `async/await` is aware of synchronization contexts. You're updating the user interface, which can be done only in a UI thread, despite also starting and waiting for long-running operations. Before `async/await`, this would've been complex and error prone.

I don't claim that `async/await` is a silver bullet for asynchrony. It doesn't magically remove all the complexity that naturally comes with the territory. Instead, it lets you focus on the inherently difficult aspects of asynchrony by taking away a lot of the boilerplate code that was previously required.

All of the features you've seen so far aim to make code simpler. The final aspect I want to mention is slightly different.

### 1.1.5 *Balancing efficiency and complexity*

I remember my first experiences with Java; it was entirely interpreted and painfully slow. After a while, optional just-in-time (JIT) compilers became available, and eventually it was taken almost for granted that any Java implementation would be JIT-compiled.

Making Java perform well took a lot of effort. This effort wouldn't have happened if the language had been a flop. But developers saw the potential and already felt more productive than they had before. Speed of development and delivery can often be more important than application speed.

C# was in a slightly different situation. The Common Language Runtime (CLR) was pretty efficient right from the start. The language support for easy interop with native code and for performance-sensitive unsafe code with pointers helps, too. C# performance continues to improve over time. (I note with a wry smile that Microsoft is now introducing tiered JIT compilation broadly like the Java HotSpot JIT compiler.)

But different workloads have different performance demands. As you'll see in section 1.2, C# is now in use across a surprising variety of platforms, including gaming and microservices, both of which can have difficult performance requirements.

Asynchrony helps address performance in some situations, but C# 7 is the most overtly performance-sensitive release. Read-only structs and a much larger surface area for `ref` features help to avoid redundant copying. The `Span<T>` feature present in modern frameworks and supported by `ref`-like struct types reduces unnecessary allocation and garbage collection. The hope is clearly that when used carefully, these techniques will cater to the requirements of specific developers.

I have a slight sense of unease around these features, as they still feel complex to me. I can't reason about a method using an `in` parameter as clearly as I can about



regular value parameters, and I'm sure it will take a while before I'm comfortable with what I can and can't do with ref locals and ref returns.

My hope is that these features will be used in moderation. They'll simplify code in situations that benefit from them, and they will no doubt be welcomed by the developers who maintain that code. I look forward to experimenting with these features in personal projects and becoming more comfortable with the balance between improved performance and increased code complexity.

I don't want to sound this note of caution too loudly. I suspect the C# team made the right choice to include the new features regardless of how much or little I'll use them in my work. I just want to point out that you don't have to use a feature just because it's there. Make your decision to opt into complexity a conscious one. Speaking of opting in, C# 7 brought a new meta-feature to the table: the use of minor version numbers for the first time since C# 1.

### 1.1.6 *Evolution at speed: Using minor versions*

The set of version numbers for C# is an odd one, and it is complicated by the fact that many developers get understandably confused between the framework and the language. (There's no C# 3.5, for example. The .NET Framework version 3.0 shipped with C# 2, and .NET 3.5 shipped with C# 3.) C# 1 had two releases: C# 1.0 and C# 1.2. Between C# 2 and C# 6 inclusive, there were only major versions that were usually backed by a new version of Visual Studio.

C# 7 bucked that trend: there were releases of C# 7.0, C# 7.1, C# 7.2, and C# 7.3, which were all available in Visual Studio 2017. I consider it highly likely that this pattern will continue in C# 8. The aim is to allow new features to evolve quickly with user feedback. The majority of C# 7.1–7.3 features have been tweaks or extensions to the features introduced in C# 7.0.

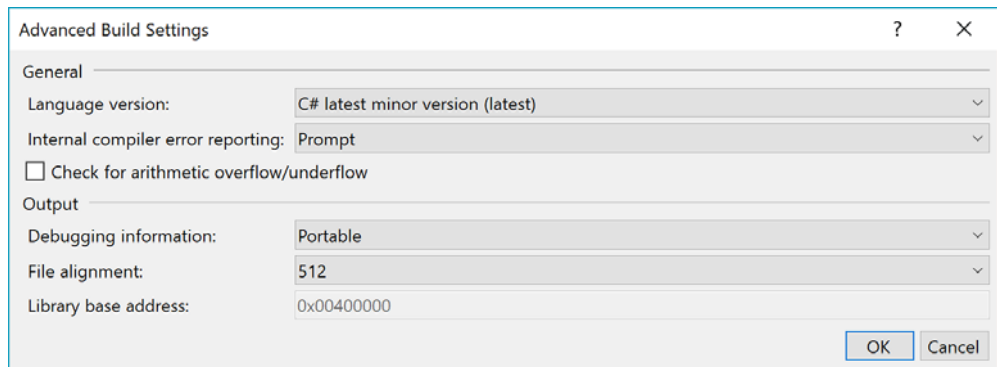
Volatility in language features can be disconcerting, particularly in large organizations. A lot of infrastructure may need to be changed or upgraded to make sure the new language version is fully supported. A lot of developers may learn and adopt new features at different paces. If nothing else, it can be a little uncomfortable for the language to change more often than you're used to.

For this reason, the C# compiler defaults to using the earliest minor version of the latest major version it supports. If you use a C# 7 compiler and don't specify any language version, it will restrict you to C# 7.0 by default. If you want to use a later minor version, you need to specify that in your project file and opt into the new features. You can do this in two ways, although they have the same effect. You can edit your project file directly to add a `<LangVersion>` element in a `<PropertyGroup>`, like this:

```
<PropertyGroup>
  ...
  <LangVersion>latest</LangVersion>
</PropertyGroup>
```

Other properties

Specifies the language version of the project



**Figure 1.1** Language version settings in Visual Studio

If you don't like editing project files directly, you can go to the project properties in Visual Studio, select the Build tab, and then click the Advanced button at the bottom right. The Advanced Build Settings dialog box, shown in figure 1.1, will open to allow you to select the language version you wish to use and other options.

This option in the dialog box isn't new, but you're more likely to want to use it now than in previous versions. The values you can select are as follows:

- *default*—The first release of the latest major version
- *latest*—The latest version
- *A specific version number*—For example, 7.0 or 7.3

This doesn't change the version of the compiler you run; it changes the set of language features available to you. If you try to use something that isn't available in the version you're targeting, the compiler error message will usually explain which version is required for that feature. If you try to use a language feature that's entirely unknown to the compiler (using C# 7 features with a C# 6 compiler, for example), the error message is usually less clear.

C# as a language has come a long way since its first release. What about the platform it runs on?

## 1.2 An evolving platform

The last few years have been exhilarating for .NET developers. A certain amount of frustration exists as well, as both Microsoft and the .NET community come to terms with the implications of a more open development model. But the overall result of the hard work by so many people is remarkable.

For many years, running C# code would almost always mean running on Windows. It would usually mean either a client-side app written in Windows Forms or Windows Presentation Foundation (WPF) or a server-side app written with ASP.NET and probably running behind Internet Information Server (IIS). Other options have been

available for a long time, and the Mono project in particular has a rich history, but the mainstream of .NET development was still on Windows.

As I write this in June 2018, the .NET world is very different. The most prominent development is .NET Core, a runtime and framework that is portable and open source, is fully supported by Microsoft on multiple operating systems, and has streamlined development tooling. Only a few years ago, that would've been unthinkable. Add to that a portable and open source IDE in the form of Visual Studio Code, and you get a flourishing .NET ecosystem with developers working on all kinds of local platforms and then deploying to all kinds of server platforms.

It would be a mistake to focus too heavily on .NET Core and ignore the many other ways C# runs these days. Xamarin provides a rich multiplatform mobile experience. Its GUI framework (Xamarin Forms) allows developers to create user interfaces that are fairly uniform across different devices where that's appropriate but that can take advantage of the underlying platform, too.

Unity is one of the most popular game-development platforms in the world. With a customized Mono runtime and ahead-of-time compilation, it can provide challenges to C# developers who are used to more-traditional runtime environments. But for many developers, this is their first or perhaps their only experience with the language.

These widely adopted platforms are far from the only ones making C#. I've recently been working with Try .NET and Blazor for very different forms of browser/C# interaction.

Try .NET allows users to write code in a browser, with autocompletion, and then build and run that code. It's great for experimenting with C# with a barrier to entry that's about as low as it can be.

Blazor is a platform for running Razor pages directly in a browser. These aren't pages rendered by a server and then displayed in the browser; the user-interface code runs within the browser using a version of the Mono runtime converted into Web-Assembly. The idea of a whole runtime executing Intermediate Language (IL) via the JavaScript engine in a browser, not only on full computers but also on mobile phones, would've struck me as absurd just a few years ago. I'm glad other developers have more imagination. A lot of the innovation in this space has been made possible only by a more collaborative and open community than ever before.

### **1.3** *An evolving community*

I've been involved in the C# community since the C# 1.0 days, and I've never seen it as vibrant as it is today. When I started using C#, it was very much seen as an "enterprise" programming language, and there was relatively little sense of fun and exploration.<sup>1</sup> With that background, the open source C# ecosystem grew fairly slowly compared with other languages, including Java, which was also considered an enterprise

---

<sup>1</sup> Don't get me wrong; it was a pleasant community to be part of, and there have always been people experimenting with C# for fun.

language. Around the time of C# 3, the alt.NET community was looking beyond the mainstream of .NET development, and this was seen as being against Microsoft in some senses.

In 2010, the NuGet (initially NuPack) package manager was launched, which made it much easier to produce and consume class libraries, whether commercial or open source. Even though the barrier of downloading a zip file, copying a DLL into somewhere appropriate, and then adding a reference to it doesn't sound hugely significant, every point of friction can put developers off.

**NOTE** Package managers other than NuGet were developed even earlier, and the OpenWrap project developed by Sebastien Lamba was particularly influential.

Fast-forward to 2014, and Microsoft announced that its Roslyn compiler platform was going to become open source under the umbrella of the new .NET Foundation. Then .NET Core was announced under the initial codename Project K; DNX came later, followed by the .NET Core tooling that's now released and stable. Then came ASP.NET Core. And Entity Framework Core. And Visual Studio Code. The list of products that truly live and breathe on GitHub goes on.

The technology has been important, but the new embrace of open source by Microsoft has been equally vital for a healthy community. Third-party open source packages have blossomed, including innovative uses for Roslyn and integrations within .NET Core tooling that just feel right.

None of this has happened in a vacuum. The rise of cloud computing makes .NET Core even more important to the .NET ecosystem than it would've been otherwise; support for Linux isn't optional. But because .NET Core is available, there's now nothing special about packaging up an ASP.NET Core service in a Docker image, deploying it with Kubernetes, and using it as just one part of a larger application that could involve many languages. The cross-pollination of good ideas between many communities has always been present, but it is stronger than ever right now.

You can learn C# in a browser. You can run C# anywhere. You can ask questions about C# on Stack Overflow and myriad other sites. You can join in the discussion about the future of the language on the C# team's GitHub repository. It's not perfect; we still have collective work to do in order to make the C# community as welcoming as it possibly can be for everyone, but we're in a great place already.

I'd like to think that *C# in Depth* has its own small place in the C# community, too. How has this book evolved?

## 1.4 An evolving book

You're reading the fourth edition of *C# in Depth*. Although the book hasn't evolved at the same pace as the language, platform, or community, it also has changed. This section will help you understand what is covered in this book.

### 1.4.1 *Mixed-level coverage*

The first edition of *C# in Depth* came out in April 2008, which was coincidentally the same time that I joined Google. Back then, I was aware that a lot of developers knew C# 1 fairly well, but they were picking up C# 2 and C# 3 as they went along without a firm grasp of how all the pieces fit together. I aimed to address that gap by diving into the language at a depth that would help readers understand not only what each feature did but why it was designed that way.

Over time, the needs of developers change. It seems to me that the community has absorbed a deeper understanding of the language almost by osmosis, at least for earlier versions. Attaining deeper understanding of the language won't be a universal experience, but for the fourth edition, I wanted the emphasis to be on the newer versions. I still think it's useful to understand the evolution of the language version by version, but there's less need to look at every detail of the features in C# 2–4.

**NOTE** Looking at the language one version at a time isn't the best way to learn the language from scratch, but it's useful if you want to understand it deeply. I wouldn't use the same structure to write a book for C# beginners.

I'm also not keen on thick books. I don't want *C# in Depth* to be intimidating, hard to hold, or hard to write in. Keeping 400 pages of coverage for C# 2–4 just didn't feel right. For that reason, I've compressed my coverage of those versions. Every feature is mentioned, and I go into detail where I feel it's appropriate, but there's less depth than in the third edition. Use the coverage in the fourth edition as a review of topics you already know and to help you determine topics you want to read more about in the third edition. You can find a link to access an electronic copy of the third edition at [www.manning.com/books/c-sharp-in-depth-fourth-edition](http://www.manning.com/books/c-sharp-in-depth-fourth-edition). Versions 5–7 of the language are covered in more detail in this edition. Asynchrony is still a tough topic to understand, and the third edition obviously doesn't cover C# 6 or 7 at all.

Writing, like software engineering, is often a balancing act. I hope the balance I've struck between detail and brevity works for you.

**TIP** If you have a physical copy of this book, I strongly encourage you to write in it. Make note of places where you disagree or parts that are particularly useful. The act of doing this will reinforce the content in your memory, and the notes will serve as reminders later.

### 1.4.2 *Examples using Noda Time*

Most of the examples I provide in the book are standalone. But to make a more compelling case for some features, it's useful to be able to point to where I use them in production code. Most of the time, I use Noda Time for this.

Noda Time is an open source project I started in 2009 to provide a better date and time library for .NET. It serves a secondary purpose, though: it's a great sandbox

project for me. It helps me hone my API design skills, learn more about performance and benchmarking, and test new C# features. All of this without breaking users, of course.

Every new version of C# has introduced features that I've been able to use in Noda Time, so I think it makes sense to use those as concrete examples in this book. All of the code is available on GitHub, which means you can clone it and experiment for yourself. The purpose of using Noda Time in examples isn't to persuade you to use the library, but I'm not going to complain if that happens to be a side effect.

In the rest of the book, I'll assume that you know what I'm talking about when I refer to Noda Time. In terms of making it suitable for examples, the important aspects of it are as follows:

- The code needs to be as readable as possible. If a language feature lets me refactor for readability, I'll jump at the chance.
- Noda Time follows semantic versioning, and new major versions are rare. I pay attention to the backward-compatibility aspects of applying new language features.
- I don't have concrete performance goals, because Noda Time can be used in many contexts with different requirements. I do pay attention to performance and will embrace features that improve efficiency, so long as they don't make the code much more complex.

To find out more about the project and check out its source code, visit <https://nodatime.org> or <https://github.com/nodatime/nodatime>.

### 1.4.3 Terminology choices

I've tried to follow the official C# terminology as closely as I can within the book, but occasionally I've allowed clarity to take precedence over precision. For example, when writing about asynchrony, I often refer to *async methods* when the same information also applies to asynchronous anonymous functions. Likewise, object initializers apply to accessible fields as well as properties, but it's simpler to mention that once and then refer only to properties within the rest of the explanation.

Sometimes the terms within the specification are rarely used in the wider community. For example, the specification has the notion of a *function member*. That's a method, property, event, indexer, user-defined operator, instance constructor, static constructor, or finalizer. It's a term for any type member that can contain executable code, and it's useful when describing language features. It's not nearly as useful when you're looking at your own code, which is why you may never have heard of it before. I've tried to use terms like this sparingly, but my view is that it's worth becoming somewhat familiar with them in the spirit of getting closer to the language.

Finally, some concepts don't have any official terminology but are still useful to refer to in a shorthand form. The one I'll use most often is probably *unspeakable names*.

This term, coined by Eric Lippert, refers to an identifier generated by the compiler to implement features such as iterator blocks or lambda expressions.<sup>2</sup> The identifier is valid for the CLR but not valid in C#; it's a name that can't be "spoken" within the language, so it's guaranteed not to clash with your code.

### **Summary**

I love C#. It's both comfortable and exciting, and I love seeing where it's going next. I hope this chapter has passed on some of that excitement to you. But this has been only a taste. Let's get onto the real business of the book without further delay.

---

<sup>2</sup> We think it was Eric, anyway. Eric can't remember for sure and thinks Anders Hejlsberg may have come up with the term first. I'll always associate it with Eric, though, along with his classification for exceptions: fatal, boneheaded, vexing, or exogenous.

## *Part 2*

### *C# 2–5*

**T**his part of the book covers all the features introduced between C# 2 (shipped with Visual Studio 2005) and C# 5 (shipped with Visual Studio 2012). This is the same set of features that took up the entire third edition of this book. Much of it feels like ancient history now; for example, we simply take it for granted that C# includes generics.

This was a tremendously productive period for C#. Some of the features I'll cover in this part are generics, nullable value types, anonymous methods, method group conversions, iterators, partial types, static classes, automatically implemented properties, implicitly typed local variables, implicitly typed arrays, object initializers, collection initializers, anonymous types, lambda expressions, extension methods, query expressions, dynamic typing, optional parameters, named arguments, COM improvements, generic covariance and contravariance, `async/await`, and caller information attributes. Phew!

I expect most of you to be at least somewhat familiar with most of the features, so I ramp up pretty fast in this part. Likewise, for the sake of reasonable brevity, I haven't gone into as much detail as I did in the third edition. The intention is to cover a variety of reader needs:

- An introduction to features you may have missed along the way
- A reminder of the features you once knew about but have forgotten
- An explanation of the reasons behind the features: why they were introduced and why they were designed in the way they were
- A quick reference in case you know what you want to do but have forgotten some syntax



If you want more detail, please refer to the third edition. As a reminder, purchase of the fourth edition entitles you to an e-book copy of the third edition.

There's one exception to this brief coverage rule: I've completely rewritten the coverage of `async/await`, which is the largest feature in C# 5. Chapter 5 covers what you need to know to use `async/await`, and chapter 6 addresses how it's implemented behind the scenes. If you're new to `async/await`, you'll almost certainly want to wait until you've used it a bit before you read chapter 6, and even then, you shouldn't expect it to be a simple read. I've tried to explain things as accessibly as I can, but the topic is fundamentally complex. I do encourage you to try, though; understanding `async/await` at a deep level can help boost your confidence when using the feature, even if you never need to dive into the IL the compiler generates for your own code. The good news is that after chapter 6, you'll find a little relief in the form of chapter 7. It's the shortest chapter in the book and a chance to recover before exploring C# 6.

With all introductions out of the way, brace yourself for an onslaught of features.

***This chapter covers***

- Using generic types and methods for flexible, safe code
- Expressing the absence of information with nullable value types
- Constructing delegates relatively easily
- Implementing iterators without writing boilerplate code

If your experience with C# goes far enough back, this chapter will be a reminder of just how far we've come and a prompt to be grateful for a dedicated and smart language design team. If you've never programmed C# without generics, you may end up wondering how C# ever took off without these features.<sup>1</sup> Either way, you may still find features you weren't aware of or details you've never considered listed here.

It's been more than 10 years since C# 2 was released (with Visual Studio 2005), so it can be hard to get excited about features in the rearview mirror. You shouldn't

---

<sup>1</sup> For me, the answer to this one is simple: C# 1 was a more productive language for many developers than Java was at the time.