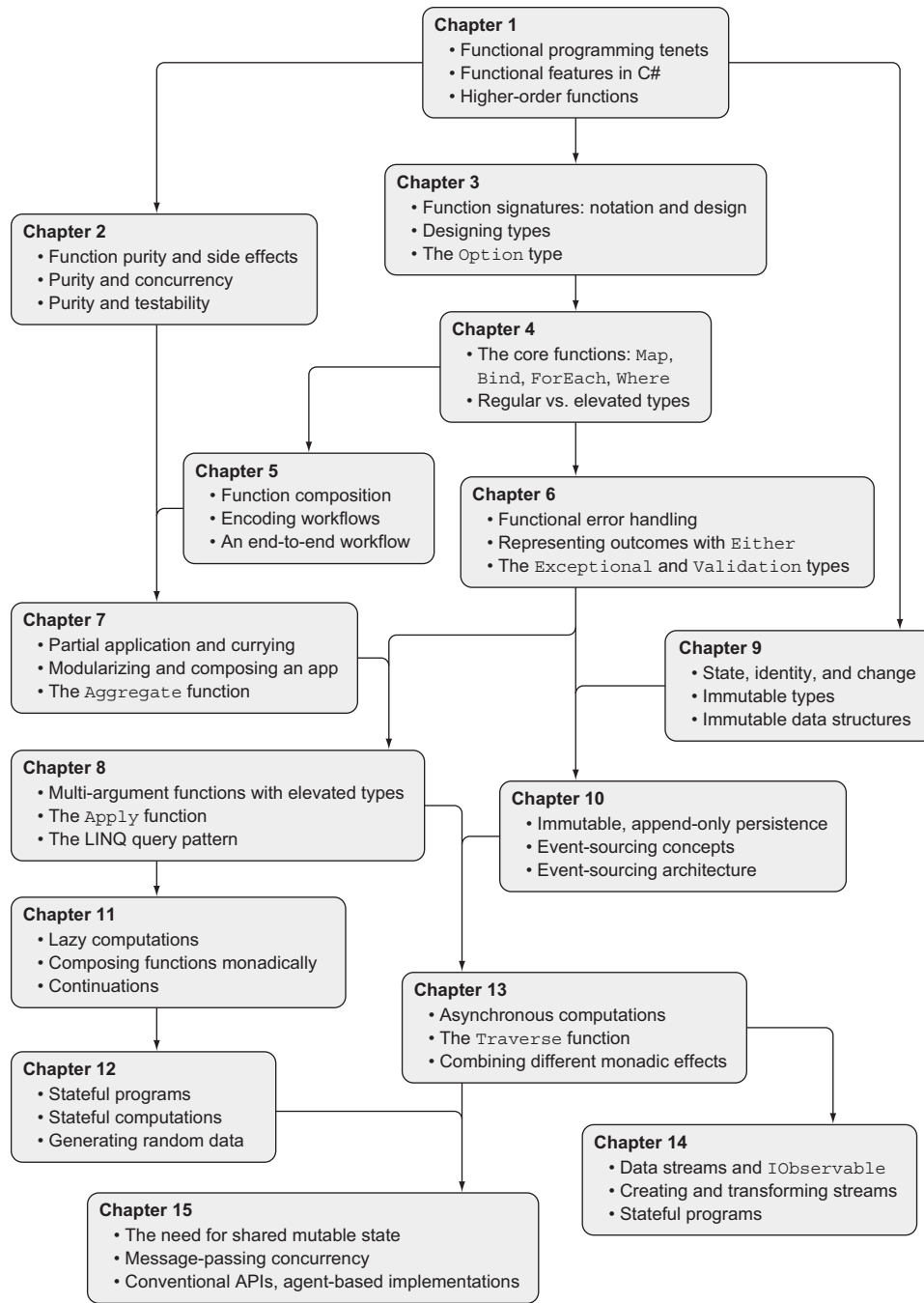


Functional Programming in

How to write better C# code

Enrico Buonanno

Inverted chapter dependency graph



Functional Programming in C#

Functional Programming in C#

ENRICO BUONANNO



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 761
Shelter Island, NY 11964
Email: orders@manning.com

©2018 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: Marina Michaels
Technical development editor: Joel Kotarski
Review editor: Aleksandar Dragosavljević
Project editor: Kevin Sullivan
Copyeditor: Andy Carroll
Proofreader: Melody Dolab
Technical proofreaders: Paul Louth, Jürgen Hoetzel
Typesetter: Gordan Salinovic
Cover designer: Leslie Haimes
Cartoons: Visoslav Radović,
Richard Sheppard
Graphic illustrations: Chuck Larson

ISBN 9781617293955

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – EBM – 22 21 20 19 18 17

To the little monkey...

brief contents

PART 1 CORE CONCEPTS1

- 1 ■ Introducing functional programming 3
- 2 ■ Why function purity matters 31
- 3 ■ Designing function signatures and types 52
- 4 ■ Patterns in functional programming 80
- 5 ■ Designing programs with function composition 102

PART 2 BECOMING FUNCTIONAL121

- 6 ■ Functional error handling 123
- 7 ■ Structuring an application with functions 149
- 8 ■ Working effectively with multi-argument functions 177
- 9 ■ Thinking about data functionally 202
- 10 ■ Event sourcing: a functional approach to persistence 229

PART 3 ADVANCED TECHNIQUES.....255

- 11 ■ Lazy computations, continuations, and the beauty of monadic composition 257
- 12 ■ Stateful programs and stateful computations 279

13	■	Working with asynchronous computations	295
14	■	Data streams and the Reactive Extensions	320
15	■	An introduction to message-passing concurrency	345

contents

preface xvii
acknowledgments xix
about this book xx

PART 1 CORE CONCEPTS1

1 *Introducing functional programming* 3

- 1.1 What is this thing called functional programming? 4
Functions as first-class values 4 ▪ *Avoiding state mutation* 5
Writing programs with strong guarantees 6
- 1.2 How functional a language is C#? 9
The functional nature of LINQ 10 ▪ *Functional features in C# 6*
and C# 7 11 ▪ *A more functional future for C#?* 13
- 1.3 Thinking in functions 14
Functions as maps 14 ▪ *Representing functions in C#* 15
- 1.4 Higher-order functions 19
Functions that depend on other functions 19 ▪ *Adapter*
functions 21 ▪ *Functions that create other functions* 22

- 1.5 Using HOFs to avoid duplication 23
 - Encapsulating setup and teardown into a HOF* 25
 - *Turning the using statement into a HOF* 26
 - *Tradeoffs of HOFs* 27
- 1.6 Benefits of functional programming 29

2 *Why function purity matters* 31

- 2.1 What is function purity? 32
 - Purity and side effects* 32
 - *Strategies for managing side effects* 33
- 2.2 Purity and concurrency 35
 - Pure functions parallelize well* 36
 - *Parallelizing impure functions* 38
 - *Avoiding state mutation* 39
- 2.3 Purity and testability 41
 - In practice: a validation scenario* 41
 - *Bringing impure functions under test* 43
 - *Why testing impure functions is hard* 45
 - *Parameterized unit tests* 46
 - *Avoiding header interfaces* 47
- 2.4 Purity and the evolution of computing 50

3 *Designing function signatures and types* 52

- 3.1 Function signature design 53
 - Arrow notation* 53
 - *How informative is a signature?* 54
- 3.2 Capturing data with data objects 55
 - Primitive types are often not specific enough* 56
 - *Constraining inputs with custom types* 57
 - *Writing “honest” functions* 59
 - *Composing values with tuples and objects* 60
- 3.3 Modeling the absence of data with Unit 61
 - Why void isn’t ideal* 61
 - *Bridging the gap between Action and Func with Unit* 63
- 3.4 Modeling the possible absence of data with Option 65
 - The bad APIs you use every day* 65
 - *An introduction to the Option type* 66
 - *Implementing Option* 68
 - *Gaining robustness by using Option instead of null* 72
 - *Option as the natural result type of partial functions* 73

4 *Patterns in functional programming* 80

- 4.1 Applying a function to a structure’s inner values 81
 - Mapping a function onto a sequence* 81
 - *Mapping a function onto an Option* 82
 - *How Option raises the level of abstraction* 84
 - *Introducing functors* 85

- 4.2 Performing side effects with ForEach 86
- 4.3 Chaining functions with Bind 88
 - Combining Option-returning functions* 89
 - Flattening nested lists with Bind* 90
 - Actually, it's called a monad* 91
 - The Return function* 92
 - Relation between functors and monads* 92
- 4.4 Filtering values with Where 93
- 4.5 Combining Option and IEnumerable with Bind 94
- 4.6 Coding at different levels of abstraction 96
 - Regular vs. elevated values* 96
 - Crossing levels of abstraction* 97
 - Map vs. Bind, revisited* 98
 - Working at the right level of abstraction* 99

5 *Designing programs with function composition* 102

- 5.1 Function composition 103
 - Brushing up on function composition* 103
 - Method chaining* 104
 - Composition in the elevated world* 104
- 5.2 Thinking in terms of data flow 105
 - Using LINQ's composable API* 105
 - Writing functions that compose well* 107
- 5.3 Programming workflows 108
 - A simple workflow for validation* 109
 - Refactoring with data flow in mind* 110
 - Composition leads to greater flexibility* 111
- 5.4 An introduction to functional domain modeling 112
- 5.5 An end-to-end server-side workflow 114
 - Expressions vs. statements* 115
 - Declarative vs. imperative* 116
 - The functional take on layering* 117

PART 2 BECOMING FUNCTIONAL.....121

6 *Functional error handling* 123

- 6.1 A safer way to represent outcomes 124
 - Capturing error details with Either* 124
 - Core functions for working with Either* 128
 - Comparing Option and Either* 129
- 6.2 Chaining operations that may fail 130
- 6.3 Validation: a perfect use case for Either 132
 - Choosing a suitable representation for errors* 132
 - Defining an Either-based API* 134
 - Adding validation logic* 134

- 6.4 Representing outcomes to client applications 136
 - Exposing an Option-like interface* 137
 - *Exposing an Either-like interface* 138
 - *Returning a result DTO* 139
- 6.5 Variations on the Either theme 140
 - Changing between different error representations* 141
 - *Specialized versions of Either* 142
 - *Refactoring to Validation and Exceptional* 143
 - *Leaving exceptions behind?* 146

7 Structuring an application with functions 149

- 7.1 Partial application: supplying arguments piecemeal 150
 - Manually enabling partial application* 152
 - *Generalizing partial application* 153
 - *Order of arguments matters* 154
- 7.2 Overcoming the quirks of method resolution 155
- 7.3 Curried functions: optimized for partial application 157
- 7.4 Creating a partial-application-friendly API 159
 - Types as documentation* 161
 - *Particularizing the data access function* 162
- 7.5 Modularizing and composing an application 164
 - Modularity in OOP* 165
 - *Modularity in FP* 167
 - *Comparing the two approaches* 169
 - *Composing the application* 170
- 7.6 Reducing a list to a single value 171
 - LINQ's Aggregate method* 171
 - *Aggregating validation results* 173
 - *Harvesting validation errors* 174

8 Working effectively with multi-argument functions 177

- 8.1 Function application in the elevated world 178
 - Understanding applicatives* 180
 - *Lifting functions* 182
 - An introduction to property-based testing* 183
- 8.2 Functors, applicatives, monads 185
- 8.3 The monad laws 187
 - Right identity* 187
 - *Left identity* 188
 - *Associativity* 189
 - Using Bind with multi-argument functions* 190
- 8.4 Improving readability by using LINQ with any monad 190
 - Using LINQ with arbitrary functors* 191
 - *Using LINQ with arbitrary monads* 192
 - *let, where, and other LINQ clauses* 195
- 8.5 When to use Bind vs. Apply 197
 - Validation with smart constructors* 197
 - *Harvesting errors with the applicative flow* 198
 - *Failing fast with the monadic flow* 199

9 *Thinking about data functionally* 202

- 9.1 The pitfalls of state mutation 203
- 9.2 Understanding state, identity, and change 206
 - Some things never change* 206
 - *Representing change without mutation* 208
- 9.3 Enforcing immutability 211
 - Immutable all the way down* 213
 - *Copy methods without boilerplate?* 214
 - *Leveraging F# for data types* 216
 - Comparing strategies for immutability: an ugly contest* 217
- 9.4 A short introduction to functional data structures 218
 - The classic functional linked list* 219
 - *Binary trees* 223

10 *Event sourcing: a functional approach to persistence* 229

- 10.1 Thinking functionally about data storage 230
 - Why data storage should be append-only* 230
 - *Relax, and forget about storing state* 231
- 10.2 Event sourcing basics 232
 - Representing events* 233
 - *Persisting events* 233
 - *Representing state* 234
 - *An interlude on pattern matching* 235
 - Representing state transitions* 238
 - *Reconstructing the current state from past events* 240
- 10.3 Architecture of an event-sourced system 241
 - Handling commands* 242
 - *Handling events* 245
 - *Adding validation* 246
 - *Creating views of the data from events* 248
- 10.4 Comparing different approaches to immutable storage 251
 - Datomic vs. Event Store* 252
 - *How event-driven is your domain?* 252

PART 3 ADVANCED TECHNIQUES255

11 *Lazy computations, continuations, and the beauty of monadic composition* 257

- 11.1 The virtue of laziness 258
 - Lazy APIs for working with Option* 259
 - *Composing lazy computations* 261
- 11.2 Exception handling with Try 263
 - Representing computations that may fail* 263
 - *Safely extracting information from a JSON object* 264
 - *Composing computations that may fail* 266
 - *Monadic composition: what does it mean?* 267

- 11.3 Creating a middleware pipeline for DB access 268
 - Composing functions that perform setup/teardown* 268
 - *A recipe against the pyramid of doom* 270
 - *Capturing the essence of a middleware function* 270
 - *Implementing the query pattern for middleware* 272
 - *Adding middleware that times the operation* 275
 - Adding middleware that manages a DB transaction* 276

12 *Stateful programs and stateful computations* 279

- 12.1 Programs that manage state 280
 - Maintaining a cache of retrieved resources* 281
 - *Refactoring for testability and error handling* 283
 - *Stateful computations* 285
- 12.2 A language for generating random data 285
 - Generating random integers* 287
 - *Generating other primitives* 287
 - *Generating complex structures* 289
- 12.3 A general pattern for stateful computations 291

13 *Working with asynchronous computations* 295

- 13.1 Asynchronous computations 296
 - The need for asynchrony* 296
 - *Representing asynchronous operations with Task* 297
 - *Task as a container for a future value* 298
 - *Handling failure* 300
 - *An HTTP API for currency conversion* 302
 - *If it fails, try a few more times* 303
 - Running asynchronous operations in parallel* 304
- 13.2 Traversables: working with lists of elevated values 306
 - Validating a list of values with monadic Traverse* 307
 - Harvesting validation errors with applicative Traverse* 309
 - Applying multiple validators to a single value* 311
 - *Using Traverse with Task to await multiple results* 312
 - *Defining Traverse for single-value structures* 313
- 13.3 Combining asynchrony and validation (or any other two monadic effects) 315
 - The problem of stacked monads* 315
 - *Reducing the number of effects* 316
 - *LINQ expressions with a monad stack* 318

14 *Data streams and the Reactive Extensions* 320

- 14.1 Representing data streams with IObservable 321
 - A sequence of values in time* 321
 - *Subscribing to an IObservable* 322
- 14.2 Creating IObservables 324
 - Creating a timer* 324
 - *Using Subject to tell an IObservable when it should signal* 325
 - *Creating IObservables from callback-based subscriptions* 326
 - *Creating IObservables from simpler structures* 327

- 14.3 Transforming and combining data streams 328
 - Stream transformations* 328
 - *Combining and partitioning streams* 330
 - *Error handling with IObservable* 332
 - Putting it all together* 334
 - 14.4 Implementing logic that spans multiple events 335
 - Detecting sequences of pressed keys* 336
 - *Reacting to multiple event sources* 338
 - *Notifying when an account becomes overdrawn* 340
 - 14.5 When should you use IObservable? 343
- 15 *An introduction to message-passing concurrency* 345
- 15.1 The need for shared mutable state 346
 - 15.2 Understanding message-passing concurrency 347
 - Implementing agents in C#* 349
 - *Getting started with agents* 351
 - *Using agents to handle concurrent requests* 352
 - Agents vs. actors* 356
 - 15.3 Functional APIs, agent-based implementations 358
 - Agents as implementation details* 358
 - *Hiding agents behind a conventional API* 360
 - 15.4 Message-passing concurrency in LOB applications 361
 - Using an agent to synchronize access to account data* 362
 - Keeping a registry of accounts* 363
 - *An agent is not an object* 364
 - *Putting it all together* 367
- Epilogue: what next?* 371
- index* 373

preface

Today, functional programming (FP) is no longer brooding in the research departments of universities; it has become an important and exciting part of mainstream programming. The majority of the languages and frameworks created in the last decade are functional, leading some to predict that the future of programming is functional. Meanwhile, popular object-oriented languages like C# and Java see the introduction of more functional features with every new release, enabling a multiparadigm programming style.

And yet, adoption in the C# community has been slow. Why is this so? One reason, I believe, is the lack of good literature:

- Most FP literature is written in and for functional languages, especially Haskell. For developers with a background in OOP, this poses a programming-language barrier to learning the concepts. Even though many of the concepts apply to a multiparadigm language like C#, learning a new paradigm *and* a new language at once is a tall order.
- Even more importantly, most of the books in the literature tend to illustrate functional techniques and concepts with examples from the domains of mathematics or computer science. For the majority of programmers who work on line-of-business (LOB) applications day in and day out, this creates a domain gap and leaves them wondering how relevant these techniques may be for real-world applications.



- Lituz.com

Elektron kitoblar

**To'liq qismini Shu tugmani
bosish orqali sotib oling!**