

# *brief contents*

---

<b>PART 1</b>	<b>FOUNDATIONS .....</b>	<b>1</b>
	1 ■ Clojure philosophy	3
	2 ■ Drinking from the Clojure fire hose	25
	3 ■ Dipping your toes in the pool	51
<b>PART 2</b>	<b>DATA TYPES .....</b>	<b>67</b>
	4 ■ On scalars	69
	5 ■ Collection types	84
<b>PART 3</b>	<b>FUNCTIONAL PROGRAMMING .....</b>	<b>115</b>
	6 ■ Being lazy and set in your ways	117
	7 ■ Functional programming	136
<b>PART 4</b>	<b>LARGE-SCALE DESIGN.....</b>	<b>171</b>
	8 ■ Macros	173
	9 ■ Combining data and code	194
	10 ■ Mutation and concurrency	224
	11 ■ Parallelism	262

<b>PART 5</b>	<b>HOST SYMBIOSIS .....</b>	<b>275</b>
	12 ■ Java.next	277
	13 ■ Why ClojureScript?	310
<b>PART 6</b>	<b>TANGENTIAL CONSIDERATIONS .....</b>	<b>331</b>
	14 ■ Data-oriented programming	333
	15 ■ Performance	363
	16 ■ Thinking programs	393
	17 ■ Clojure changes the way you think	423

# contents

---

*foreword to the second edition* xix  
*foreword to the first edition* xxi  
*preface* xxiii  
*acknowledgments* xxv  
*about this book* xxvii  
*about clojure* xxxvii  
*about the cover illustration* xxxix

## PAT 1 FOUNDATIONS .....1

### **1** Clojure philosophy 3

#### 1.1 The Clojure way 4

*Simplicity* 4 ▪ *Freedom to focus* 5 ▪ *Empowerment* 5  
*Clarity* 6 ▪ *Consistency* 7

#### 1.2 Why a(nother) Lisp? 8

*Beauty* 9 ▪ *But what's with all the parentheses?* 9

#### 1.3 Functional programming 16

*A workable definition of functional programming* 16  
*The implications of functional programming* 17

#### 1.4 Why Clojure isn't especially object-oriented 17

*Defining terms* 17 ▪ *Imperative "baked in"* 18 ▪ *Most of what OOP gives you, Clojure provides* 19

#### 1.5 Summary 24

## 2 *Drinking from the Clojure fire hose* 25

- 2.1 Scalars: the base data types 26
  - Numbers* 26 ▪ *Integers* 27 ▪ *Floating-point numbers* 27
  - Rationals* 28 ▪ *Symbols* 28 ▪ *Keywords* 28 ▪ *Strings* 29
  - Characters* 29
- 2.2 Putting things together: collections 29
  - Lists* 29 ▪ *Vectors* 30 ▪ *Maps* 30 ▪ *Sets* 30
- 2.3 Making things happen: calling functions 31
- 2.4 Vars are not variables 31
- 2.5 Functions 32
  - Anonymous functions* 32 ▪ *Creating named functions with def and defn* 33 ▪ *Functions with multiple arities* 33 ▪ *In-place functions with #()* 34
- 2.6 Locals, loops, and blocks 35
  - Blocks* 35 ▪ *Locals* 35 ▪ *Loops* 36
- 2.7 Preventing things from happening: quoting 39
  - Evaluation* 39 ▪ *Quoting* 40 ▪ *Unquote* 41 ▪ *Unquote-splicing* 42 ▪ *Auto-gensym* 42
- 2.8 Using host libraries via interop 43
  - Accessing static class members (Clojure only)* 43 ▪ *Creating instances* 43 ▪ *Accessing instance members with the . operator* 44 ▪ *Setting instance fields* 44 ▪ *The .. macro* 44
  - The doto macro* 45 ▪ *Defining classes* 45
- 2.9 Exceptional circumstances 46
  - Throwing and catching* 46
- 2.10 Modularizing code with namespaces 47
  - Creating namespaces using ns* 47 ▪ *Loading other namespaces with :require* 48 ▪ *Loading and creating mappings with :refer* 48 ▪ *Creating mappings with :refer* 49 ▪ *Loading Java classes with :import* 49
- 2.11 Summary 50

## 3 *Dipping your toes in the pool* 51

- 3.1 Truthiness 52
  - What's truth?* 52 ▪ *Don't create Boolean objects* 52
  - nil vs. false* 53
- 3.2 Nil pun with care 53

- 3.3 Destructuring 55
  - Your assignment, should you choose to accept it* 55
  - Destructuring with a vector* 56 ▪ *Destructuring with a map* 57
  - Destructuring in function parameters* 59 ▪ *Destructuring vs. accessor methods* 59
- 3.4 Using the REPL to experiment 59
  - Experimenting with seqs* 59 ▪ *Experimenting with graphics* 61
  - Putting it all together* 63 ▪ *When things go wrong* 63 ▪ *Just for fun* 65
- 3.5 Summary 66

## PART 2 DATA TYPES ..... 67

### 4 On scalars 69

- 4.1 Understanding precision 70
  - Truncation* 70 ▪ *Promotion* 71 ▪ *Overflow* 71
  - Underflow* 72 ▪ *Rounding errors* 72
- 4.2 Trying to be rational 73
  - Why be rational?* 73 ▪ *How to be rational* 74 ▪ *Caveats of rationality* 75
- 4.3 When to use keywords 75
  - Applications of keywords* 76 ▪ *Qualifying your keywords* 77
- 4.4 Symbolic resolution 78
  - Metadata* 79 ▪ *Symbols and namespaces* 80 ▪ *Lisp-1* 80
- 4.5 Regular expressions—the second problem 81
  - Syntax* 82 ▪ *Regular-expression functions* 83 ▪ *Beware of mutable matchers* 83
- 4.6 Summary 83

### 5 Collection types 84

- 5.1 Persistence, sequences, and complexity 85
  - “You keep using that word. I do not think it means what you think it means.”* 85 ▪ *Sequence terms and what they mean* 86
  - Big-O* 89
- 5.2 Vectors: creating and using them in all their varieties 91
  - Building vectors* 91 ▪ *Large vectors* 92 ▪ *Vectors as stacks* 95
  - Using vectors instead of reverse* 96 ▪ *Subvectors* 97 ▪ *Vectors as map entries* 97 ▪ *What vectors aren’t* 98

- 5.3 Lists: Clojure's code-form data structure 99
  - Lists like Lisps like* 99 ▪ *Lists as stacks* 100 ▪ *What lists aren't* 100
- 5.4 How to use persistent queues 101
  - A queue about nothing* 101 ▪ *Putting things on* 102
  - Getting things* 102 ▪ *Taking things off* 102
- 5.5 Persistent sets 103
  - Basic properties of Clojure sets* 103 ▪ *Keeping your sets in order with sorted-set* 104 ▪ *The contains? function* 105 ▪ *The clojure.set namespace* 105
- 5.6 Thinking in maps 107
  - Hash maps* 107 ▪ *Keeping your keys in order with sorted maps* 108 ▪ *Keeping your insertions in order with array maps* 109
- 5.7 Putting it all together: finding the position of items in a sequence 110
  - Implementation* 111
- 5.8 Summary 113

## PART 3 FUNCTIONAL PROGRAMMING ..... 115

### 6 *Being lazy and set in your ways* 117

- 6.1 On immutability: being set in your ways 117
  - What is immutability?* 118 ▪ *What is immutability for?* 119
- 6.2 Structural sharing: a persistent toy 120
- 6.3 Laziness 123
  - Familiar laziness with logical-and* 124 ▪ *Understanding the lazy-seq recipe* 125 ▪ *Losing your head* 128 ▪ *Employing infinite sequences* 129 ▪ *The delay and force macros* 130
- 6.4 Putting it all together: a lazy quicksort 132
  - The implementation* 133
- 6.5 Summary 135

### 7 *Functional programming* 136

- 7.1 Functions in all their forms 136
  - First-class functions* 137 ▪ *Higher-order functions* 140 ▪ *Pure functions* 144 ▪ *Named arguments* 145 ▪ *Constraining functions with pre- and postconditions* 146

- 7.2 On closures 148
  - Functions returning closures* 149 ▪ *Closing over parameters* 150
  - Passing closures as functions* 150 ▪ *Sharing closure context* 151
- 7.3 Thinking recursively 155
  - Mundane recursion* 155 ▪ *Tail calls and recur* 158 ▪ *Don't forget your trampoline* 161 ▪ *Continuation-passing style* 163
- 7.4 Putting it all together: A\* pathfinding 165
  - The world* 165 ▪ *Neighbors* 165 ▪ *The A\* implementation* 167 ▪ *Notes about the A\* implementation* 169
- 7.5 Summary 170

## PART 4 LARGE-SCALE DESIGN.....171

### 8 *Macros* 173

- 8.1 Data is code is data 175
  - Syntax-quote, unquote, and splicing* 176 ▪ *Macro rules of thumb* 177
- 8.2 Defining control structures 178
  - Defining control structures without syntax-quote* 178 ▪ *Defining control structures using syntax-quote and unquoting* 179
- 8.3 Macros combining forms 180
- 8.4 Using macros to change forms 182
- 8.5 Using macros to control symbolic resolution time 186
  - Anaphora* 186 ▪ *(Arguably) useful selective name capturing* 188
- 8.6 Using macros to manage resources 188
- 8.7 Putting it all together: macros returning functions 190
- 8.8 Summary 193

### 9 *Combining data and code* 194

- 9.1 Namespaces 195
  - Creating namespaces* 196 ▪ *Expose only what's needed* 197
  - Declarative inclusions and exclusions* 199
- 9.2 Exploring Clojure multimethods with the Universal Design Pattern 200
  - The parts* 201 ▪ *Basic use of the Universal Design Pattern* 202
  - Multimethods to the rescue* 203 ▪ *Ad hoc hierarchies for inherited*

- behaviors* 203 ▪ *Resolving conflict in hierarchies* 204
- Arbitrary dispatch for true maximum power* 205
- 9.3 Types, protocols, and records 206
  - Records* 206 ▪ *Protocols* 209 ▪ *Building from a more primitive base with deftype* 217
- 9.4 Putting it all together: a fluent builder for chess moves 219
  - Java implementation* 219 ▪ *Clojure implementation* 221
- 9.5 Summary 223

## 10 Mutation and concurrency 224

- 10.1 When to use refs 226
  - Using refs for a mutable game board* 228 ▪ *Transactions* 230
  - Embedded transactions* 232 ▪ *The things that STM makes easy* 232 ▪ *Potential downsides* 233 ▪ *The things that make STM unhappy* 234
- 10.2 Refactoring with refs 235
  - Fixing the game board example* 235 ▪ *Commutative change with commute* 237 ▪ *Vulgar change with ref-set* 238 ▪ *Refs under stress* 239
- 10.3 When to use agents 240
  - In-process vs. distributed concurrency models* 241 ▪ *Controlling I/O with an agent* 243 ▪ *The difference between send and send-off* 245 ▪ *Error handling* 246 ▪ *When not to use agents* 248
- 10.4 When to use atoms 249
  - Sharing across threads* 249 ▪ *Using atoms in transactions* 250
- 10.5 When to use locks 252
  - Safe mutation through locking* 253 ▪ *Using Java's explicit locks* 254
- 10.6 Vars and dynamic binding 256
  - The binding macro* 257 ▪ *Creating a named var* 257
  - Creating anonymous vars* 258 ▪ *Dynamic scope* 259
- 10.7 Summary 260

## 11 Parallelism 262

- 11.1 When to use futures 263
  - Futures as callbacks* 263



- 11.2 When to use promises 268
  - Parallel tasks with promises* 269
  - Callback API to blocking API* 270
  - Deterministic deadlocks* 271
- 11.3 Parallel operations 271
  - The pvalues macro* 272
  - The pmap function* 272
  - The pcalls function* 273
- 11.4 A brief introduction to reducer/fold 273
- 11.5 Summary 274

## PART 5 HOST SYMBIOSIS ..... 275

### 12 *Java.next* 277

- 12.1 Generating objects on the fly with proxy 278
  - A simple dynamic web service* 279
- 12.2 Clojure gen-class and GUI programming 285
  - Namespaces as class specifications* 286
  - The guts of namespace compilation* 288
  - Exploring user interface design and development with Clojure* 289
- 12.3 Clojure's relationship to Java arrays 292
  - Types of arrays: primitive and reference* 292
  - Array mutability* 294
  - Arrays' unfortunate naming convention* 295
  - Multidimensional arrays* 296
  - Variadic method/constructor calls* 297
- 12.4 All Clojure functions implement ... 297
  - The java.util.Comparator interface* 297
  - The java.lang.Runnable interface* 298
  - The java.util.concurrent.Callable interface* 299
- 12.5 Using Clojure data structures in Java APIs 299
  - The java.util.List interface* 300
  - The java.lang.Comparable interface* 300
  - The java.util.RandomAccess interface* 301
  - The java.util.Collection interface* 301
  - The java.util.Set interface* 302
- 12.6 The definterface macro 302
  - Generating interfaces on the fly* 302
- 12.7 Be wary of exceptions 304
  - A bit of background regarding exceptions* 305
  - Runtime vs. compile-time exceptions* 305
  - Handling exceptions* 307
  - Custom exceptions* 308
- 12.8 Summary 309

## 13 *Why ClojureScript?* 310

- 13.1 Implementation vs. interface 311
- 13.2 Compiler internals: analysis vs. emission 314
  - Stages of compilation* 315 ▪ *Web Audio* 317 ▪ *Advanced compilation* 321 ▪ *Generating an externs.js file* 324
- 13.3 Compile vs. run 326
- 13.4 Summary 330

## PART 6 TANGENTIAL CONSIDERATIONS .....331

## 14 *Data-oriented programming* 333

- 14.1 Code as code, and data as data 334
  - A strict line betwixt* 334 ▪ *ORMG* 335 ▪ *Common ways to derive information from data* 337 ▪ *PLOP* 337
- 14.2 Data as data 338
  - The benefits of value* 338 ▪ *Tagged literals* 343
- 14.3 Data as code 347
  - The data-programmable engine* 347 ▪ *Examples of data-programmable engines* 347 ▪ *Case study: simple event sourcing* 348
- 14.4 Code as data as code 357
  - Hart's discovery and homoiconicity* 358 ▪ *Clojure code is data* 358 ▪ *Putting parentheses around the specification* 358
- 14.5 Summary 362

## 15 *Performance* 363

- 15.1 Type hints 364
  - Advantages of type adornment* 364 ▪ *Type-hinting arguments and returns* 364 ▪ *Type-hinting objects* 366
- 15.2 Transients 366
  - Ephemeral garbage* 366 ▪ *Transients compare in efficiency to mutable collections* 367
- 15.3 Chunked sequences 368
  - Regaining one-at-a-time laziness* 370

- 15.4 Memoization 370
  - Reexamining memoization* 371 ▪ *A memoization protocol* 371
  - Abstraction-oriented programming* 373
- 15.5 Understanding coercion 374
  - Using primitive longs* 375 ▪ *Using primitive doubles* 376
  - Using auto-promotion* 377
- 15.6 Reducibles 378
  - An example reducible collection* 379 ▪ *Deriving your first reducing function transformer* 380 ▪ *More reducing function transformers* 383 ▪ *Reducible transformers* 385 ▪ *Performance of reducibles* 386 ▪ *Drawbacks of reducibles* 387
  - Integrating reducibles with Clojure reduce* 387 ▪ *The fold function: reducing in parallel* 389
- 15.7 Summary 392

## 16 *Thinking programs* 393

- 16.1 A problem of search 394
  - A brute-force Sudoku solver* 394 ▪ *Declarative is the goal* 399
- 16.2 Thinking data via unification 400
  - Potential equality, or satisfiability* 400 ▪ *Substitution* 404
  - Unification* 405
- 16.3 An introduction to core.logic 407
  - It's all about unification* 407 ▪ *Relations* 408
  - Subgoals* 411
- 16.4 Constraints 414
  - An introduction to constraint programming* 414 ▪ *Limiting binding via finite domains* 416 ▪ *Solving Sudoku with finite domains* 418
- 16.5 Summary 421

## 17 *Clojure changes the way you think* 423

- 17.1 Thinking in the domain 424
  - A ubiquitous DSL* 424 ▪ *Implementing a SQL-like DSL to generate queries* 426 ▪ *A note about Clojure's approach to DSLs* 432

- 17.2 Testing 432
  - Some useful unit-testing techniques* 433 ■ *Contracts programming* 435
- 17.3 Invisible design patterns 437
  - Clojure's first-class design patterns* 437
- 17.4 Error handling and debugging 447
  - Error handling* 447 ■ *Debugging* 450
- 17.5 Fare thee well 454
  - resources* 455
  - index* 461