CLOJURE FOR THE BRAVE AND TRUE

learn the ultimate language and become a better programmer

Daniel Higginbotham

no starch press

CLOJURE FOR THE BRAVE AND TRUE

CLOJURE FOR THE BRAVE AND TRUE

learn the ultimate language and become a better programmer

Daniel Higginbotham

no starch press

CLOJURE FOR THE BRAVE AND TRUE. Copyright © 2015 by Daniel Higginbotham.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

Printed in USA

First printing

19 18 17 16 15 1 2 3 4 5 6 7 8 9

ISBN-10: 1-59327-591-9
ISBN-13: 978-1-59327-591-4

Publisher: William Pollock
Production Editor: Riley Hoffman
Cover Design: Beth Middleworth and Daniel and Jessica Higginbotham
Cover and Interior Illustrations: Jessica Higginbotham
Interior Design: Octopod Studios
Developmental Editors: Hayley Baker and Seph Kramer
Technical Reviewer: Alan Dipert
Copyeditor: Anne Marie Walker
Compositors: Riley Hoffman and Susan Glinert Stevens
Proofreader: Emelie Burnette

For information on distribution, translations, or bulk sales, please contact No Starch Press, Inc. directly:

No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
phone: 415.863.9900; info@nostarch.com
www.nostarch.com

Library of Congress Cataloging-in-Publication Data

Higginbotham, Daniel.
Clojure for the brave and true : learn the ultimate language and become a better programmer / by Daniel Higginbotham.
pages cm
Includes index.
Summary: "Guide to the functional programming language Clojure. Teaches tools and techniques for writing programs in Clojure. Covers how to wield and compose Clojure's core functions; use Emacs for Clojure development; write macros to modify the Clojure programming language; and use Clojure's tools to simplify concurrency and parallel programming"-- Provided by publisher.
ISBN 978-1-59327-591-4 -- ISBN 1-59327-591-9
1. Clojure (Computer program language) I. Title.
QA76.73.C565H54 2015
005.13'3--dc23
2015014205

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

About the Author

Daniel Higginbotham has been a professional programmer for 11 years, half of that at McKinsey & Company, where he used Clojure to build mobile and web applications. He has also contributed to the curriculum for ClojureBridge, an organization that offers free, beginner-friendly Clojure workshops for women. Daniel blogs about life and programming at *http://flyingmachinestudios.com/*, and can be found on Twitter, @nonrecursive. He lives in Durham, North Carolina, with his wife and four cats.

About the Technical Reviewer

Alan Dipert first heard about Lisp when he was 10 years old. After it was described to him, he said "That sounds dumb." In 2009, he learned Clojure and revised his opinion. Alan has designed and built Clojure systems, conducted Clojure trainings, and spoken at Clojure conferences. You can keep track of Alan's work and recent opinions by visiting *http://tailrecursion.com/~alan* or by following him on Twitter, @alandipert.

For Jess

BRIEF CONTENTS

Foreword by Alan Dipert . . . xvii

Acknowledgments . . . xix

Introduction . . . xxi

PART I: ENVIRONMENT SETUP

Chapter 1: Building, Running, and the REPL . . . 3

Chapter 2: How to Use Emacs, an Excellent Clojure Editor . . . 11

PART II: LANGUAGE FUNDAMENTALS

Chapter 3: Do Things: A Clojure Crash Course . . . 35

Chapter 4: Core Functions in Depth . . . 71

Chapter 5: Functional Programming . . . 97

Chapter 6: Organizing Your Project: A Librarian's Tale . . . 125

Chapter 7: Clojure Alchemy: Reading, Evaluation, and Macros . . . 147

Chapter 8: Writing Macros . . . 165

PART III: ADVANCED TOPICS

Chapter 9: The Sacred Art of Concurrent and Parallel Programming . . . 189

Chapter 10: Clojure Metaphysics: Atoms, Refs, Vars, and Cuddle Zombies . . . 207

Chapter 11: Mastering Concurrent Processes with core.async . . . 233

Chapter 12: Working with the JVM . . . 247

Chapter 13: Creating and Extending Abstractions with Multimethods, Protocols, and Records . . . 265

Appendix A: Building and Developing with Leiningen. 277

Appendix B: Boot, the Fancy Clojure Build Framework 281

Farewell!. 291

Index 293

CONTENTS IN DETAIL

FOREWORD by Alan Dipert **xvii**

ACKNOWLEDGMENTS **xix**

INTRODUCTION **xxi**

Learning a New Programming Language: A Journey Through the Four Labyrinths. xxii
How This Book Is Organized xxii
Part I: Environment Setup xxii
Part II: Language Fundamentals. xxiii
Part III: Advanced Topics xxiv
The Code xxv
The Journey Begins! xxv

PART I: ENVIRONMENT SETUP

1
BUILDING, RUNNING, AND THE REPL **3**

First Things First: What Is Clojure? 4
Leiningen 5
Creating a New Clojure Project 5
Running the Clojure Project 6
Building the Clojure Project 7
Using the REPL 7
Clojure Editors. 9
Summary 9

2
HOW TO USE EMACS, AN EXCELLENT CLOJURE EDITOR **11**

Installation. 12
Configuration 13
Emacs Escape Hatch 14
Emacs Buffers 14
Working with Files 15
Key Bindings and Modes 17
Emacs Is a Lisp Interpreter 17
Modes. 18
Installing Packages 19
Core Editing Terminology and Key Bindings 19
Point 20
Movement 20
Selection with Regions 20
Killing and the Kill Ring 21
Editing and Help. 22
Using Emacs with Clojure 23
Fire Up Your REPL! 23
Interlude: Emacs Windows and Frames 24

A Cornucopia of Useful Key Bindings . . . 25
How to Handle Errors . . . 27
Paredit . . . 28
Continue Learning . . . 30
Summary . . . 31

PART II: LANGUAGE FUNDAMENTALS

3
DO THINGS: A CLOJURE CRASH COURSE 35

Syntax . . . 36
Forms . . . 36
Control Flow . . . 37
Naming Values with def . . . 40
Data Structures . . . 41
Numbers . . . 42
Strings . . . 42
Maps . . . 43
Keywords . . . 44
Vectors . . . 45
Lists . . . 45
Sets . . . 46
Simplicity . . . 48
Functions . . . 48
Calling Functions . . . 48
Function Calls, Macro Calls, and Special Forms . . . 50
Defining Functions . . . 51
Anonymous Functions . . . 57
Returning Functions . . . 58
Pulling It All Together . . . 59
The Shire's Next Top Model . . . 59
let . . . 61
loop . . . 63
Regular Expressions . . . 64
Symmetrizer . . . 65
Better Symmetrizer with reduce . . . 66
Hobbit Violence . . . 67
Summary . . . 69
Exercises . . . 69

4
CORE FUNCTIONS IN DEPTH 71

Programming to Abstractions . . . 72
Treating Lists, Vectors, Sets, and Maps as Sequences . . . 73
first, rest, and cons . . . 74
Abstraction Through Indirection . . . 77
Seq Function Examples . . . 79
map . . . 79
reduce . . . 80
take, drop, take-while, and drop-while . . . 81

filter and some . . . 83
sort and sort-by . . . 84
concat . . . 84
Lazy Seqs . . . 84
Demonstrating Lazy Seq Efficiency . . . 84
Infinite Sequences . . . 87
The Collection Abstraction . . . 88
into . . . 88
conj . . . 90
Function Functions . . . 90
apply . . . 91
partial . . . 91
complement . . . 92
A Vampire Data Analysis Program for the FWPD . . . 93
Summary . . . 96
Exercises . . . 96

5
FUNCTIONAL PROGRAMMING 97

Pure Functions: What and Why . . . 98
Pure Functions Are Referentially Transparent . . . 98
Pure Functions Have No Side Effects . . . 99
Living with Immutable Data Structures . . . 100
Recursion Instead of for/while . . . 100
Function Composition Instead of Attribute Mutation . . . 103
Cool Things to Do with Pure Functions . . . 105
comp . . . 105
memoize . . . 107
Peg Thing . . . 108
Playing . . . 108
Code Organization . . . 110
Creating the Board . . . 111
Moving Pegs . . . 117
Rendering and Printing the Board . . . 120
Player Interaction . . . 121
Summary . . . 124
Exercises . . . 124

6
ORGANIZING YOUR PROJECT: A LIBRARIAN'S TALE 125

Your Project as a Library . . . 126
Storing Objects with def . . . 127
Creating and Switching to Namespaces . . . 129
refer . . . 130
alias . . . 132
Real Project Organization . . . 133
The Relationship Between File Paths and Namespace Names . . . 133
Requiring and Using Namespaces . . . 134
The ns Macro . . . 138
To Catch a Burglar . . . 140
Summary . . . 144

7
CLOJURE ALCHEMY: READING, EVALUATION, AND MACROS 147

An Overview of Clojure's Evaluation Model . . . 148
The Reader . . . 153
Reading . . . 153
Reader Macros . . . 154
The Evaluator . . . 155
These Things Evaluate to Themselves . . . 156
Symbols . . . 156
Lists . . . 159
Macros . . . 160
Syntactic Abstraction and the -> Macro . . . 163
Summary . . . 164
Exercises . . . 164

8
WRITING MACROS 165

Macros Are Essential . . . 166
Anatomy of a Macro . . . 167
Building Lists for Evaluation . . . 168
Distinguishing Symbols and Values . . . 168
Simple Quoting . . . 169
Syntax Quoting . . . 171
Using Syntax Quoting in a Macro . . . 173
Refactoring a Macro and Unquote Splicing . . . 174
Things to Watch Out For . . . 176
Variable Capture . . . 176
Double Evaluation . . . 178
Macros All the Way Down . . . 179
Brews for the Brave and True . . . 180
Validation Functions . . . 180
if-valid . . . 182
Summary . . . 184
Exercises . . . 184

PART III: ADVANCED TOPICS

9
THE SACRED ART OF CONCURRENT AND PARALLEL PROGRAMMING 189

Concurrency and Parallelism Concepts . . . 190
Managing Multiple Tasks vs. Executing Tasks Simultaneously . . . 190
Blocking and Asynchronous Tasks . . . 191
Concurrent Programming and Parallel Programming . . . 191
Clojure Implementation: JVM Threads . . . 191
What's a Thread? . . . 192
The Three Goblins: Reference Cells, Mutual Exclusion, and Dwarven Berserkers . . . 193

Futures, Delays, and Promises . . . 196
Futures . . . 196
Delays . . . 198
Promises . . . 200
Rolling Your Own Queue . . . 202
Summary . . . 205
Exercises . . . 206

10
CLOJURE METAPHYSICS: ATOMS, REFS, VARS, AND CUDDLE ZOMBIES 207

Object-Oriented Metaphysics . . . 208
Clojure Metaphysics . . . 210
Atoms . . . 212
Watches and Validators . . . 215
Watches . . . 215
Validators . . . 217
Refs . . . 218
Modeling Sock Transfers . . . 218
commute . . . 221
Vars . . . 223
Dynamic Binding . . . 223
Altering the Var Root . . . 227
Stateless Concurrency and Parallelism with pmap . . . 228
Summary . . . 232
Exercises . . . 232

11
MASTERING CONCURRENT PROCESSES WITH CORE.ASYNC 233

Getting Started with Processes . . . 234
Buffering . . . 236
Blocking and Parking . . . 237
thread . . . 238
The Hot Dog Machine Process You've Been Longing For . . . 239
alts!! . . . 241
Queues . . . 243
Escape Callback Hell with Process Pipelines . . . 244
Additional Resources . . . 245
Summary . . . 245

12
WORKING WITH THE JVM 247

The JVM . . . 248
Writing, Compiling, and Running a Java Program . . . 250
Object-Oriented Programming in the World's Tiniest Nutshell . . . 250
Ahoy, World . . . 251
Packages and Imports . . . 253
JAR Files . . . 255
clojure.jar . . . 255
Clojure App JARs . . . 257

Java Interop . . . 257
Interop Syntax . . . 258
Creating and Mutating Objects . . . 259
Importing . . . 260
Commonly Used Java Classes . . . 261
The System Class . . . 261
The Date Class . . . 262
Files and Input/Output . . . 262
Resources . . . 264
Summary . . . 264

13
CREATING AND EXTENDING ABSTRACTIONS WITH MULTIMETHODS, PROTOCOLS, AND RECORDS 265

Polymorphism . . . 266
Multimethods . . . 266
Protocols . . . 269
Records . . . 272
Further Study . . . 275
Summary . . . 275
Exercises . . . 275

A
BUILDING AND DEVELOPING WITH LEININGEN 277

The Artifact Ecosystem . . . 277
Identification . . . 278
Dependencies . . . 278
Plug-Ins . . . 279
Summary . . . 280

B
BOOT, THE FANCY CLOJURE BUILD FRAMEWORK 281

Boot's Abstractions . . . 282
Tasks . . . 282
The REPL . . . 284
Composition and Coordination . . . 285
Handlers and Middleware . . . 285
Tasks Are Middleware Factories . . . 287
Filesets . . . 288
Next Steps . . . 289

FAREWELL! 291

INDEX 293

FOREWORD

As you read this hilarious book, you will at some point experience a very serious moment. It is the moment you admit to yourself that programming is more enjoyable after knowing some Clojure. It is also the moment that your investment in this book, in money and time, comes back to you—with interest.

Humor has a certain relationship to seriousness. It is appropriate to joke about serious things, but only after the right amount of time has passed. For example, it took years for me to be able to crack a smile when I remember my favorite uncle's last words: "Hold my beer."

This book works in the opposite way. It points out really funny things for the right amount of time *before*, and perhaps even *during*, the serious event—that moment you realize you enjoy programming more because of Clojure. It does this without obscuring the deep, technical aspects of Clojure programming that you will learn.

This approach is refreshing because most of the programming books I've read are drier than a camel's fart. We are fortunate that Daniel is a brilliant programmer and writer and that his wife Jess is an equally brilliant illustrator. We are especially fortunate that they both went insane and decided to write a book at exactly the same time.

Clojure is the topic of this book, but in a way it—or perhaps its creator, Rich Hickey—is also one of the authors, since Clojure is the most elegant

programming language ever designed. Like the concept of brunch, Clojure is so elegant that it's difficult to tell anyone anything about it without somehow improving them.

Elegance is a quality regularly ascribed to many dialects in the family of programming languages known collectively as Lisp, of which Clojure is one. All Lisps descend from a set of simple and beautiful discoveries made by the mathematician John McCarthy in 1958.

Since 1958, there have been many Lisps and Lisp books. There are many more Lisps and books to come. As artifacts of the past and future, each are right for the unique combination of constraints and desires faced and fancied by their authors, in their respective times.

I find Clojure, and this particular book about it, especially right for the present. I hope you will too.

Alan Dipert

ACKNOWLEDGMENTS

So many people helped me birth this weird baby, and I am humbled and grateful for all their support.

First, thanks to Jess, my wife, for doing the illustrations that complete this book, giving it the visual character I had hoped for. Thanks, too, for the support and for putting up with me when I was in crazy-eyed writer mode. (P.S. It feels bizarre to thank my wife via a programming book's front matter.)

Thanks to my friends and colleagues at McKinsey who read early revisions and encouraged me to keep writing. Foremost among them are Pat Shaughnessy, Alex Rothenberg, Thomas Newton, Jalil Fanaian, Chris Parker, Mark Daggett, Christian Lilley, and Mike Morreale. Y'all are so great; please move to Durham.

Thanks to my friend Bridget Hillyer for being a constant source of support and positivity. I always feel like you have my back, and it means a lot to me! Thanks, too, to my friend Joe Jackson, for reading, listening to me blather, and offering feedback, and for making me feel cool by gushing about this book to other people in front of me. Alan Dipert, friend, tech reviewer, and now coworker, I give thee a million thanks for your excellent technical editing and for introducing me to Clojure in the first place.

I don't know if every writer continually asks himself, "Why the hell am I doing this? Is anyone even going to read it?" but I certainly did. So I want to

thank all the friendly folks who wrote to me and suggested edits while the initial web version of this book was being written. That positive feedback made me feel confident that I was doing something worthwhile. By the same token, thanks to everyone who bought the Leanpub version!

A big thank you to Clojure community leaders Eric Normand, David Nolen, and Alex Miller for giving this book positive press. See you at the next Conj!

Finally, a bazillion thank yous to the magnificent folks at No Starch Press for all your help in shaping this book into something I'm immensely proud of. Thank you for your high standards. Thank you for continually pushing for greater clarity, and for even suggesting jokes. (Seph's "ball of wax" edit still makes me crack up.) Seph Kramer, Riley Hoffman, Hayley Baker, Alison Law, Tyler Ortman, Anne Marie Walker: thank you!

INTRODUCTION

Deep in your innermost being, you've always known you were destined to learn Clojure. Every time you held your keyboard aloft, crying out in anguish over an incomprehensible class hierarchy; every time you lay awake at night, disturbing your loved ones with sobs over a mutation-induced heisenbug; every time a race condition caused you to pull out more of your ever-dwindling hair, some secret part of you has known that *there has to be a better way*.

Now, at long last, the instructional material you have in front of your face will unite you with the programming language you've been longing for.

Learning a New Programming Language: A Journey Through the Four Labyrinths

To wield Clojure to its fullest, you'll need to find your way through the four labyrinths that face every programmer learning a new language:

The Forest of Tooling A friendly and efficient programming environment makes it easy to try your ideas. You'll learn how to set up your environment.

The Mountain of Language As you ascend, you'll gain knowledge of Clojure's syntax, semantics, and data structures. You'll learn how to use one of the mightiest programming tools, the macro, and learn how to simplify your life with Clojure's concurrency constructs.

The Cave of Artifacts In its depths you'll learn to build, run, and distribute your own programs, and how to use code libraries. You'll also learn Clojure's relationship to the Java Virtual Machine (JVM).

The Cloud Castle of Mindset In its rarefied air, you'll come to know the why and how of Lisp and functional programming. You'll learn about the philosophy of simplicity that permeates Clojure, and how to solve problems like a Clojurist.

Make no mistake, you will work. But this book will make the work feel exhilarating, not exhausting. That's because this book follows three guidelines:

- It takes the dessert-first approach, giving you the development tools and language details you need to start playing with real programs immediately.
- It assumes zero experience with the JVM, functional programming, or Lisp. It covers these topics in detail so you'll feel confident about what you're doing when you build and run Clojure programs.
- It eschews *real-world* examples in favor of more interesting exercises, like *assaulting hobbits* and *tracking glittery vampires*.

By the end, you'll be able to use Clojure, one of the most exciting and fun programming languages in existence!

How This Book Is Organized

This book is split into three parts to better guide you through your valiant quest, brave fledgling Clojurist.

Part I: Environment Setup

To stay motivated and learn efficiently, you need to actually write code and build executables. These chapters take you on a quick tour of the tools you'll need to easily write programs. That way you can focus on learning Clojure, not fiddling with your environment.

Chapter 1: Building, Running, and the REPL

There's something powerful and motivating about getting a real program running. Once you can do that, you're free to experiment, and you can actually share your work!

In this short chapter, you'll invest a small amount of time to become familiar with a quick way to build and run Clojure programs. You'll learn how to experiment with code in a running Clojure process using a read-eval-print loop (REPL). This will tighten your feedback loop and help you learn more efficiently.

Chapter 2: How to Use Emacs, an Excellent Clojure Editor

A quick feedback loop is crucial for learning. In this chapter, I cover Emacs from the ground up to guarantee you have an efficient Emacs/Clojure workflow.

Part II: Language Fundamentals

These chapters give you a solid foundation on which to continue learning Clojure. You'll start by learning Clojure's basics (syntax, semantics, and data structures) so you can *do things*. Then you'll take a step back to examine Clojure's most used functions in detail and learn how to solve problems with them using the *functional programming* mindset.

Chapter 3: Do Things: A Clojure Crash Course

This is where you'll start to really dig into Clojure. It's also where you'll need to close your windows because you'll start shouting, "*HOLY MOLEY THAT'S SPIFFY!*" at the top of your lungs and won't stop until you've hit this book's index.

You've undoubtedly heard of Clojure's awesome concurrency support and other stupendous features, but Clojure's most salient characteristic is that it is a Lisp. You'll explore this Lisp core, which is composed of two parts: functions and data.

Chapter 4: Core Functions in Depth

In this chapter, you'll learn about a couple of Clojure's underlying concepts. This will give you the grounding you need to read the documentation for functions you haven't used before and to understand what's happening when you try them.

You'll also see usage examples of the functions you'll be reaching for the most. This will give you a solid foundation for writing your own code and for reading and learning from other people's projects. And remember how I mentioned tracking glittery vampires? You'll do that in this chapter (unless you already do it in your spare time).

Chapter 5: Functional Programming

In this chapter, you'll take your concrete experience with functions and data structures and integrate it with a new mindset: the functional programming mindset. You'll show off your knowledge by constructing the hottest new game that's sweeping the nation: Peg Thing!

Chapter 6: Organizing Your Project: A Librarian's Tale

This chapter explains what namespaces are and how to use them to organize your code. I don't want to give away too much, but it also involves an international cheese thief.

Chapter 7: Clojure Alchemy: Reading, Evaluation, and Macros

In this chapter, we'll take a step back and describe how Clojure runs your code. This will give you the conceptual structure you need to truly understand how Clojure works and how it's different from other, non-Lisp languages. With this structure in place, I'll introduce the macro, one of the most powerful tools in existence.

Chapter 8: Writing Macros

This chapter thoroughly examines how to write macros, starting with basic examples and advancing in complexity. You'll close by donning your make-believe cap, pretending that you run an online potion store and using macros to validate customer orders.

Part III: Advanced Topics

These chapters cover Clojure's extra-fun topics: concurrency, Java interop, and abstraction. Although you can write programs without understanding these tools and concepts, they're intellectually rewarding and give you tremendous power as a programmer. One of the reasons people say that learning Clojure makes you a better programmer is that it makes the concepts covered in these chapters easy to understand and practical to use.

Chapter 9: The Sacred Art of Concurrent and Parallel Programming

In this chapter, you'll learn what concurrency and parallelism are and why they matter. You'll learn about the challenges you'll face when writing parallel programs and about how Clojure's design helps to mitigate them. You'll use futures, delays, and promises to safely write parallel programs.

Chapter 10: Clojure Metaphysics: Atoms, Refs, Vars, and Cuddle Zombies

This chapter goes into great detail about Clojure's approach to managing state and how that simplifies concurrent programming. You'll learn how to use atoms, refs, and vars, three constructs for managing state, and you'll learn how to do stateless parallel computation with `pmap`. And there will be cuddle zombies.

Chapter 11: Mastering Concurrent Processes with core.async

In this chapter, you'll ponder the idea that everything in the universe is a hot dog vending machine. By which I mean you'll learn how to model systems of independently running processes that communicate with each other over channels using the core.async library.

Chapter 12: Working with the JVM

This chapter is like a cross between a phrase book and cultural introduction to the Land of Java. It gives you an overview of what the JVM is, how it runs programs, and how to compile programs for it. It also gives you a brief tour of frequently used Java classes and methods, and explains how to interact with them from Clojure. More than that, it shows you how to think about and understand Java so you can incorporate any Java library into your Clojure program.

Chapter 13: Creating and Extending Abstractions with Multimethods, Protocols, and Records

In Chapter 4 you learn that Clojure is written in terms of abstractions. This chapter serves as an introduction to the world of creating and implementing your own abstractions. You'll learn the basics of multimethods, protocols, and records.

Appendix A: Building and Developing with Leiningen

This appendix clarifies some of the finer points of working with Leiningen, like what Maven is and how to figure out the version numbers of Java libraries so that you can use them.

Appendix B: Boot, the Fancy Clojure Build Framework

Boot is an alternative to Leiningen that provides the same functionally, but with the added bonus that it's easier to extend and write composable tasks. This appendix explains Boot's underlying concepts and guides you through writing your first tasks.

The Code

You can download all the source code from the book at *http://www.nostarch.com/clojure/*. The code is organized by chapter.

Chapter 1 describes the different ways that you can run Clojure code, including how to use a REPL. I recommend running most of the examples in the REPL as you encounter them, especially in Chapters 3 through 8. This will help you get used to writing and understanding Lisp code, and it will help you retain everything you're learning. But for the examples that are long, it's best to write your code to a file, and then run the code you wrote in a REPL.

The Journey Begins!

Are you ready, brave reader? Are you ready to meet your true destiny? Grab your best pair of parentheses: you're about to embark on the journey of a lifetime!

PART I

ENVIRONMENT SETUP

1

BUILDING, RUNNING, AND THE REPL

In this chapter, you'll invest a small amount of time up front to get familiar with a quick, foolproof way to build and run Clojure programs. It feels great to get a real program running. Reaching that milestone frees you up to experiment, share your work, and gloat to your colleagues who are still using last decade's languages. This will help keep you motivated!

You'll also learn how to instantly run code within a running Clojure process using a *Read-Eval-Print Loop (REPL)*, which allows you to quickly test your understanding of the language and learn more efficiently.

But first, I'll briefly introduce Clojure. Next, I'll cover Leiningen, the de facto standard build tool for Clojure. By the end of the chapter, you'll know how to do the following:

- Create a new Clojure project with Leiningen
- Build the project to create an executable JAR file
- Execute the JAR file
- Execute code in a Clojure REPL

First Things First: What Is Clojure?

Clojure was forged in a mythic volcano by Rich Hickey. Using an alloy of Lisp, functional programming, and a lock of his own epic hair, he crafted a language that's delightful yet powerful. Its Lisp heritage gives you the power to write code more expressively than is possible in most non-Lisp languages, and its distinct take on functional programming will sharpen your thinking as a programmer. Plus, Clojure gives you better tools for tackling complex domains (like concurrent programming) that are traditionally known to drive developers into years of therapy.

When talking about Clojure, though, it's important to keep in mind the distinction between the Clojure language and the Clojure compiler. The Clojure language is a Lisp dialect with a functional emphasis whose syntax and semantics are independent of any implementation. The compiler is an executable JAR file, *clojure.jar*, which takes code written in the Clojure language and compiles it to Java Virtual Machine (JVM) bytecode. You'll see *Clojure* used to refer to both the language and the compiler, which can be confusing if you're not aware that they're separate things. But now that you're aware, you'll be fine.

This distinction is necessary because, unlike most programming languages like Ruby, Python, C, and a bazillion others, Clojure is a *hosted language*. Clojure programs are executed within a JVM and rely on the JVM for core features like threading and garbage collection. Clojure also targets JavaScript and the Microsoft Common Language Runtime (CLR), but this book only focuses on the JVM implementation.

We'll explore the relationship between Clojure and the JVM more later on, but for now the main concepts you need to understand are these:

- JVM processes execute Java bytecode.
- Usually, the Java Compiler produces Java bytecode from Java source code.
- JAR files are collections of Java bytecode.
- Java programs are usually distributed as JAR files.
- The Java program *clojure.jar* reads Clojure source code and produces Java bytecode.
- That Java bytecode is then executed by the same JVM process already running *clojure.jar*.

Clojure continues to evolve. As of this writing, it's at version 1.7.0, and development is going strong. If you're reading this book in the far future and Clojure has a higher version number, don't worry! This book covers Clojure's fundamentals, which shouldn't change from one version to the next. There's no need for your robot butler to return this book to the bookstore.

Now that you know what Clojure is, let's actually build a freakin' Clojure program!

Leiningen

These days, most Clojurists use Leiningen to build and manage their projects. You can read a full description of Leiningen in Appendix A, but for now we'll focus on using it for four tasks:

1. Creating a new Clojure project
2. Running the Clojure project
3. Building the Clojure project
4. Using the REPL

Before continuing, make sure you have Java version 1.6 or later installed. You can check your version by running `java -version` in your terminal, and download the latest Java Runtime Environment (JRE) from *http://www.oracle.com/technetwork/java/javase/downloads/index.html*. Then, install Leiningen using the instructions on the Leiningen home page at *http://leiningen.org/* (Windows users, note there's a Windows installer). When you install Leiningen, it automatically downloads the Clojure compiler, *clojure.jar*.

Creating a New Clojure Project

Creating a new Clojure project is very simple. A single Leiningen command creates a project skeleton. Later, you'll learn how to do tasks like incorporate Clojure libraries, but for now, these instructions will enable you to execute the code you write.

Go ahead and create your first Clojure project by typing the following in your terminal:

```
lein new app clojure-noob
```

This command should create a directory structure that looks similar to this (it's okay if there are some differences):

```
  | .gitignore
  | doc
  | | intro.md
❶ | project.clj
  | README.md
❷ | resources
  | src
```

```
  | | clojure_noob
❸ | | | core.clj
❹ | test
  | | clojure_noob
  | | | core_test.clj
```

This project skeleton isn't inherently special or Clojure-y. It's just a convention used by Leiningen. You'll be using Leiningen to build and run Clojure apps, and Leiningen expects your app to have this structure. The first file of note is *project.clj* at ❶, which is a configuration file for Leiningen. It helps Leiningen answer such questions as "What dependencies does this project have?" and "When this Clojure program runs, what function should run first?" In general, you'll save your source code in *src/<project_name>*. In this case, the file *src/clojure_noob/core.clj* at ❸ is where you'll be writing your Clojure code for a while. The *test* directory at ❹ obviously contains tests, and *resources* at ❷ is where you store assets like images.

Running the Clojure Project

Now let's actually run the project. Open *src/clojure_noob/core.clj* in your favorite editor. You should see this:

```
❶ (ns clojure-noob.core
    (:gen-class))

❷ (defn -main
    "I don't do a whole lot...yet."
    [& args]
❸   (println "Hello, World!"))
```

The lines at ❶ declare a namespace, which you don't need to worry about right now. The `-main` function at ❷ is the *entry point* to your program, a topic that is covered in Appendix A. For now, replace the text `"Hello, World!"` at ❸ with `"I'm a little teapot!"`. The full line should read `(println "I'm a little teapot!")`.

Next, navigate to the *clojure_noob* directory in your terminal and enter:

```
lein run
```

You should see the output `"I'm a little teapot!"` Congratulations, little teapot, you wrote and executed a program!

You'll learn more about what's actually happening in the program as you read through the book, but for now all you need to know is that you created a function, `-main`, and that function runs when you execute `lein run` at the command line.

Building the Clojure Project

Using `lein run` is great for trying out your code, but what if you want to share your work with people who don't have Leiningen installed? To do that, you can create a stand-alone file that anyone with Java installed (which is basically everyone) can execute. To create the file, run this:

```
lein uberjar
```

This command creates the file *target/uberjar/clojure-noob-0.1.0-SNAPSHOT-standalone.jar*. You can make Java execute it by running this:

```
java -jar target/uberjar/clojure-noob-0.1.0-SNAPSHOT-standalone.jar
```

Look at that! The file *target/uberjar/clojure-noob-0.1.0-SNAPSHOT-standalone.jar* is your new, award-winning Clojure program, which you can distribute and run on almost any platform.

You now have all the basic details you need to build, run, and distribute (very) basic Clojure programs. In later chapters, you'll learn more details about what Leiningen is doing when you run the preceding commands, gaining a complete understanding of Clojure's relationship to the JVM and how you can run production code.

Before we move on to Chapter 2 and discuss the wonder and glory of Emacs, let's go over another important tool: the REPL.

Using the REPL

The REPL is a tool for experimenting with code. It allows you to interact with a running program and quickly try out ideas. It does this by presenting you with a prompt where you can enter code. It then *reads* your input, *evaluates* it, *prints* the result, and *loops*, presenting you with a prompt again.

This process enables a quick feedback cycle that isn't possible in most other languages. I strongly recommend that you use it frequently because you'll be able to quickly check your understanding of Clojure as you learn. Besides that, REPL development is an essential part of the Lisp experience, and you'd really be missing out if you didn't use it.

To start a REPL, run this:

```
lein repl
```

The output should look like this:

```
nREPL server started on port 28925
REPL-y 0.1.10
Clojure 1.7.0
    Exit: Control+D or (exit) or (quit)
Commands: (user/help)
    Docs: (doc function-name-here)
          (find-doc "part-of-name-here")
```

```
  Source: (source function-name-here)
          (user/sourcery function-name-here)
 Javadoc: (javadoc java-object-or-class-here)
Examples from clojuredocs.org: [clojuredocs or cdoc]
          (user/clojuredocs name-here)
          (user/clojuredocs "ns-here" "name-here")
clojure-noob.core=>
```

The last line, `clojure-noob.core=>`, tells you that you're in the `clojure-noob.core` namespace. You'll learn about namespaces later, but for now notice that the namespace basically matches the name of your *src/clojure_noob/core.clj* file. Also, notice that the REPL shows the version as *Clojure 1.7.0*, but as mentioned earlier, everything will work okay no matter which version you use.

The prompt also indicates that your code is loaded in the REPL, and you can execute the functions that are defined. Right now only one function, `-main`, is defined. Go ahead and execute it now:

```
clojure-noob.core=> (-main)
I'm a little teapot!
nil
```

Well done! You just used the REPL to evaluate a function call. Try a few more basic Clojure functions:

```
clojure-noob.core=> (+ 1 2 3 4)
10
clojure-noob.core=> (* 1 2 3 4)
24
clojure-noob.core=> (first [1 2 3 4])
1
```

Awesome! You added some numbers, multiplied some numbers, and took the first element from a vector. You also had your first encounter with weird Lisp syntax! All Lisps, Clojure included, employ *prefix notation*, meaning that the operator always comes first in an expression. If you're unsure about what that means, don't worry. You'll learn all about Clojure's syntax soon.

; huh, never thought of it that way

Conceptually, the REPL is similar to Secure Shell (SSH). In the same way that you can use SSH to interact with a remote server, the Clojure REPL allows you to interact with a running Clojure process. This feature can be very powerful because you can even attach a REPL to a live production app and modify your program as it runs. For now, though, you'll be using the REPL to build your knowledge of Clojure syntax and semantics.

; i think this is what makes figwheel tick

One more note: going forward, this book will present code without REPL prompts, but please do try the code! Here's an example:

```
(do (println "no prompt here!")
  (+ 1 3))
```

```
; => no prompt here!
; => 4
```

When you see code snippets like this, lines that begin with `; =>` indicate the output of the code being run. In this case, the text `no prompt here` should be printed, and the return value of the code is `4`.

Clojure Editors

At this point you should have the basic knowledge you need to begin learning the Clojure language without having to fuss with an editor or integrated development environment (IDE). But if you do want a good tutorial on a powerful editor, Chapter 2 covers Emacs, the most popular editor among Clojurists. You absolutely do not need to use Emacs for Clojure development, but Emacs offers tight integration with the Clojure REPL and is well-suited to writing Lisp code. What's most important, however, is that you use whatever works for you.

If Emacs isn't your cup of tea, here are some resources for setting up other text editors and IDEs for Clojure development:

- This YouTube video will show you how to set up Sublime Text 2 for Clojure development: *http://www.youtube.com/watch?v=wBl0rYXQdGg/*.
- Vim has good tools for Clojure development. This article is a good starting point: *http://mybuddymichael.com/writings/writing-clojure-with-vim-in-2013.html*.
- Counterclockwise is a highly recommended Eclipse plug-in: *https://github.com/laurentpetit/ccw/wiki/GoogleCodeHome*.
- Cursive Clojure is the recommended IDE for those who use IntelliJ: *https://cursiveclojure.com/*.
- Nightcode is a simple, free IDE written in Clojure: *https://github.com/oakes/Nightcode/*.

Summary

I'm so proud of you, little teapot. You've run your first Clojure program! Not only that, but you've become acquainted with the REPL, one of the most important tools for developing Clojure software. Amazing! It brings to mind the immortal lines from "Long Live" by one of my personal heroes:

> You held your head like a hero
> On a history book page
> It was the end of a decade
> But the start of an age
> —Taylor Swift

Bravo!

2

HOW TO USE EMACS, AN EXCELLENT CLOJURE EDITOR

On your journey to Clojure mastery, your editor will be your closest ally. I highly recommend working with Emacs, but you can, of course, use any editor you want. If you don't follow the thorough Emacs instructions in this chapter, or if you choose to use a different editor, it's worthwhile to at least invest some time in setting up your editor to work with a REPL.

The reason I recommend Emacs is that it offers tight integration with a Clojure REPL, which allows you to instantly try out your code as you write. That kind of tight feedback loop will be useful while learning Clojure and, later, when writing real Clojure programs. Emacs is also great for working with any Lisp dialect; in fact, Emacs is written in a Lisp dialect called Emacs Lisp (elisp).

By the end of this chapter, your Emacs setup will look something like Figure 2-1.

Figure 2-1: A typical Emacs setup for working with Clojure—code on one side, REPL on the other

To get there, you'll start by installing Emacs and setting up a new-person-friendly Emacs configuration. Then you'll learn the basics: how to open, edit, and save files, and how to interact with Emacs using essential key bindings. Finally, you'll learn how to actually edit Clojure code and interact with the REPL.

Installation

You should use the latest major version of Emacs, Emacs 24, for the platform you're working on:

OS X Install vanilla Emacs as a Mac app from *http://emacsformacosx.com/*. Other options, like Aquamacs, are supposed to make Emacs more "Mac-like," but they're problematic in the long run because they're set up so differently from standard Emacs that it's difficult to use the Emacs manual or follow along with tutorials.

Ubuntu Follow the instructions at *https://launchpad.net/~cassou/+archive/emacs*.

Windows You can find a binary at *http://ftp.gnu.org/gnu/emacs/windows/*. After you download and unzip the latest version, you can run the Emacs executable under *bin\runemacs.exe*.

After you've installed Emacs, open it. You should see something like Figure 2-2.

Figure 2-2: The screen Emacs displays when you open it for the first time

Welcome to the cult of Emacs! You've made Richard Stallman proud!

Configuration

I've created a repository of all the files you need to configure Emacs for Clojure, available in this book's resources at *https://www.nostarch.com/clojure/*. Do the following to delete your existing Emacs configuration and install the Clojure-friendly one:

1. Close Emacs.
2. Delete *~/.emacs* or *~/.emacs.d* if they exist. This is where Emacs looks for configuration files, and deleting these files and directories will ensure that you start with a clean slate.
3. Download the Emacs configuration zip file from the book's resource page and unzip it. Its contents should be a folder, *emacs-for-clojure-book1*. Run `mv path/to/emacs-for-clojure-book1 ~/.emacs.d`.
4. Create the file *~/.lein/profiles.clj* and add this line to it:

```
{:user {:plugins [[cider/cider-nrepl "0.8.1"]]}}
```

5. Open Emacs.

When you open Emacs, you should see a lot of activity as Emacs downloads a bunch of useful packages. Once the activity stops, go ahead and just quit Emacs, and then open it again. After you do so, you should see a window like the one in Figure 2-3.

Figure 2-3: How Emacs looks after installing your sweet new configuration

Now that we've got everything set up, let's learn how to use Emacs!

Emacs Escape Hatch

Before we dig in to the fun stuff, you need to know an important Emacs key binding: CTRL-G. This key binding quits whatever Emacs command you're trying to run. So if things aren't going right, hold down CTRL, press G, and then try again. It won't close Emacs or make you lose any work; it'll just cancel your current action.

Emacs Buffers

All editing happens in an Emacs *buffer*. When you first start Emacs, a buffer named `*scratch*` is open. Emacs will always show you the name of the current buffer at the bottom of the window, as shown in Figure 2-4.

Figure 2-4: Emacs will always show you the name of the current buffer.

By default, the `*scratch*` buffer handles parentheses and indentation in a way that's optimal for Lisp development but is inconvenient for writing plain text. Let's create a fresh buffer so we can play around without having unexpected things happen. To create a buffer, do this:

1. Hold down CTRL and press X.
2. Release CTRL.
3. Press B.

We can express the same sequence in a more compact format: **C-x b**.

After performing this key sequence, you'll see a prompt at the bottom of the application, as shown in Figure 2-5.

Figure 2-5: The minibuffer is where Emacs prompts you for input.

This area is called the *minibuffer*, and it is where Emacs prompts you for input. Right now it's prompting us for a buffer name. You can enter the name of a buffer that is already open, or you can enter a new buffer name. Type in **`emacs-fun-times`** and press ENTER. You should now see a completely blank buffer and can just start typing. You'll find that keys mostly work the way you'd expect. Characters appear as you type them. The up, down, left, and right arrow keys move you as you'd expect, and ENTER creates a new line.

You'll also notice that you're not suddenly sporting a bushy Unix beard or Birkenstocks (unless you had them to begin with). This should help ease any lingering trepidation you feel about using Emacs. When you're done messing around, go ahead and *kill* the buffer by typing **C-x k enter**. (It might come as a surprise, but Emacs is actually quite violent, making ample use of the term *kill*.)

Now that you've killed the `emacs-fun-times` buffer, you should be back in the `*scratch*` buffer. In general, you can create as many new buffers as you want with **C-x b**. You can also quickly switch between buffers using the same command. When you create a new buffer this way, it exists only in memory until you save it as a file; buffers aren't necessarily backed by files, and creating a buffer doesn't necessarily create a file. Let's learn about working with files.

Working with Files

The key binding for opening a file in Emacs is **C-x C-f**. Notice that you'll need to hold down CTRL when pressing both X and F. After you do that, you'll get another minibuffer prompt. Navigate to *~/.emacs.d/customizations/ui.el*, which customizes the way Emacs looks and how you can interact with it. Emacs opens the file in a new buffer with the same name as the filename. Let's go to line 37 and uncomment it by removing the leading semicolons. It will look like this:

```
(setq initial-frame-alist '((top . 0) (left . 0) (width . 120) (height . 80)))
```

Then change the values for `width` and `height`, which set the dimensions in *characters* for the active window. By changing these values, you can set the

Emacs window to open at a certain size every time it starts. Try something small at first, like 80 and 20:

```
(setq initial-frame-alist '((top . 0) (left . 0) (width . 80) (height . 20)))
```

Now save your file with the following key binding: **C-x C-s**. You should get a message at the bottom of Emacs like `Wrote /Users/snuffleupagus/.emacs.d/customizations/ui.el`. You can also try saving your buffer using the key binding you use in other applications (for example, CTRL-S or ⌘-S). The Emacs configuration you downloaded should allow that to work, but if it doesn't, it's no big deal.

After saving the file, quit Emacs and start it again. I bet it's very tiny! See my example in Figure 2-6.

Figure 2-6: You can configure Emacs to set its height and width every time you open it.

Go through that same process a couple of times until Emacs starts at a size that you like. Or just comment out those lines again and be done with it (in which case Emacs will open at its default width and height). If you're done editing *ui.el*, you can close its buffer with **C-x k**. Either way, you're done saving your first file in Emacs! If something crazy happens, you can follow the instructions in "Configuration" on page 13 to get Emacs working again.

If you want to create a new file, just use **C-x C-f** and enter the new file's path in the minibuffer. As soon as you save the buffer, Emacs will create a file with the buffer's contents at the path you entered.

Let's recap:

1. In Emacs, editing takes place in buffers.
2. To switch to a buffer, use **C-x b** and enter the buffer name in the minibuffer.
3. To create a new buffer, use **C-x b** and enter a new buffer name.

4. To open a file, use **C-x C-f** and navigate to the file.
5. To save a buffer to a file, use **C-x C-s**.
6. To create a new file, use **C-x C-f** and enter the new file's path. When you save the buffer, Emacs will create the file on the filesystem.

Key Bindings and Modes

You've already come a long way! You can now use Emacs like a very basic editor. This should help you get by if you ever need to use Emacs on a server or are forced into pairing with an Emacs nerd.

However, to really be productive, it'll be useful for you to know some *key* details about key bindings (ha-ha!). Then I'll introduce Emacs modes. After that, I'll cover some core terminology and go over a bunch of super useful key bindings.

Emacs Is a Lisp Interpreter

The term *key binding* derives from the fact that Emacs binds *keystrokes* to *commands*, which are just elisp functions (I'll use *command* and *function* interchangeably). For example, **C-x b** is bound to the function `switch-to-buffer`. Likewise, **C-x C-s** is bound to `save-file`.

But Emacs goes even further than that. Even simple keystrokes like **f** and **a** are bound to a function, in this case `self-insert-command`, the command for adding characters to the buffer you're editing.

From Emacs's point of view, all functions are created equal, and you can redefine all functions, even core functions like `save-file`. You probably won't *want* to redefine core functions, but you can.

You can redefine functions because, at its core, Emacs is *just* a Lisp interpreter that happens to load code-editing facilities. Most of Emacs is written in elisp, so from the perspective of Emacs, `save-file` is just a function, as is `switch-to-buffer` and almost any other command you can run. Not only that, but any functions you create are treated the same way as built-in functions. You can even use Emacs to execute elisp, modifying Emacs as it runs.

The freedom to modify Emacs using a powerful programming language is what makes Emacs so flexible and why people like me are so crazy about it. Yes, it has a lot of surface-level complexity that can take time to learn. But underlying Emacs is the elegant simplicity of Lisp and the infinite tinkerability that comes with it. This tinkerability isn't limited to just creating and redefining functions. You can also create, redefine, and remove key bindings. Conceptually, key bindings are just an entry in a lookup table associating keystrokes with functions, and that lookup table is completely modifiable.

You can also run commands by name, without a specific key binding, using **M-x** ***`function-name`*** (for example, **M-x `save-buffer`**). *M* stands for *meta*, a key that modern keyboards don't possess but which is mapped to ALT on Windows and Linux and OPTION on Macs. **M-x** runs the `smex` command, which prompts you for the name of another command to be run.

Now that you understand key bindings and functions, you'll be able to understand what modes are and how they work.

Modes

An Emacs *mode* is a collection of key bindings and functions that are packaged together to help you be productive when editing different types of files. (Modes also do things like tell Emacs how to do syntax highlighting, but that's of secondary importance, and I won't cover that here.)

For example, when you're editing a Clojure file, you'll want to load Clojure mode. Right now I'm writing a Markdown file and using Markdown mode, which has lots of useful key bindings specific to working with Markdown. When editing Clojure, it's best to have a set of Clojure-specific key bindings, like **C-c C-k** to load the current buffer into a REPL and compile it.

Modes come in two flavors: *major* modes and *minor* modes. Markdown mode and Clojure mode are major modes. Major modes are usually set by Emacs when you open a file, but you can also set the mode explicitly by running the relevant Emacs command, for example with **M-x** `clojure-mode` or **M-x** `major-mode`. Only one major mode is active at a time.

Whereas major modes specialize Emacs for a certain file type or language, minor modes usually provide functionality that's useful across file types. For example, abbrev mode "automatically expands text based on pre-defined abbreviation definitions" (per the Emacs manual[1]). You can have multiple minor modes active at the same time.

You can see which modes are active on the *mode line*, as shown in Figure 2-7.

Figure 2-7: The mode line shows you which modes are active.

1. *http://www.gnu.org/software/emacs/manual/html_node/emacs/Minor-Modes.html.*

If you open a file and Emacs doesn't load a major mode for it, chances are that one exists. You'll just need to download its package. Speaking of which . . .

Installing Packages

Many modes are distributed as *packages*, which are just bundles of elisp files stored in a package repository. Emacs 24, which you installed at the beginning of this chapter, makes it very easy to browse and install packages. **M-x `package-list-packages`** will show you almost every package available; just make sure you run **M-x `package-refresh-contents`** first so you get the latest list. You can install packages with **M-x `package-install`**.

You can also customize Emacs by loading your own elisp files or files you find on the Internet. The "Beginner's Guide to Emacs" (found at *http://www.masteringemacs.org/articles/2010/10/04/beginners-guide-to-emacs/*) has a good description of how to load customizations under the section "Loading New Packages" toward the bottom of the article.

Core Editing Terminology and Key Bindings

If all you want to do is use Emacs like a text editor, you can skip this section entirely! But you'll be missing out on some great stuff. In this section, we'll go over key Emacs terms; how to select, cut, copy, and paste text; how to select, cut, copy, and paste text (see what I did there? Ha ha ha!); and how to move through the buffer efficiently.

To get started, open a new buffer in Emacs and name it *jack-handy*. Then enter the following Jack Handy quotations:

```
If you were a pirate, you know what would be the one thing that would
really make you mad? Treasure chests with no handles. How the hell are
you supposed to carry it?!

The face of a child can say it all, especially the mouth part of the
face.

To me, boxing is like a ballet, except there's no music, no
choreography, and the dancers hit each other.
```

Use this example to experiment with navigation and editing in this section.

Point

If you've been following along, you should see a red-orange rectangle in your Emacs buffer. This is the *cursor*, and it's the graphical representation of the *point*. Point is where all the magic happens: you insert text at point, and most editing commands happen in relation to point. And even though your cursor appears to rest on top of a character, point is actually located between that character and the previous one.

For example, place your cursor over the *f* in *If you were a pirate*. Point is located between *I* and *f*. Now, if you use **C-k**, all the text from the letter *f* onward will disappear. **C-k** runs the command `kill-line`, which *kills* all text after point on the current line (I'll talk more about killing later). Undo that change with **C-/**. Also, try your normal OS key binding for undo; that should work as well.

Movement

You can use your arrow keys to move point just like any other editor, but many key bindings allow you to navigate more efficiently, as shown in Table 2-1.

Table 2-1: Key Bindings for Navigating Text

Keys	Description
C-a	Move to beginning of line.
M-m	Move to first non-whitespace character on the line.
C-e	Move to end of line.
C-f	Move forward one character.
C-b	Move backward one character.
M-f	Move forward one word (I use this a lot).
M-b	Move backward one word (I use this a lot, too).
C-s	Regex search for text in current buffer and move to it. Press **C-s** again to move to next match.
C-r	Same as **C-s**, but search in reverse.
M-<	Move to beginning of buffer.
M->	Move to end of buffer.
M-g g	Go to line.

Go ahead and try out these key bindings in your *jack-handy* buffer!

Selection with Regions

In Emacs, we don't *select* text. We create *regions*, and we do so by setting the *mark* with **C-spc** (CTRL-spacebar). Then, when you move point, everything between mark and point is the region. It's very similar to SHIFT-selecting text for basic purposes.

For example, do the following in your *jack-handy* buffer:

1. Go to the beginning of the file.
2. Use **C-spc**.
3. Use **M-f** twice. You should see a highlighted region encompassing *If you*.
4. Press backspace. That should delete *If you*.

One cool thing about using mark instead of Shift-selecting text is that you're free to use all of Emacs's movement commands after you set the mark. For example, you could set a mark and then use **C-s** to search for some bit of text hundreds of lines down in your buffer. Doing so would create a very large region, and you wouldn't have to strain your pinky holding down SHIFT.

Regions also let you confine an operation to a limited area of the buffer. Try this:

1. Create a region encompassing *The face of a child can say it all*.
2. Use **M-x** `replace-string` and replace *face* with *head*.

This will perform the replacement within the current region rather than the entire buffer after point, which is the default behavior.

Killing and the Kill Ring

In most applications we can *cut* text, which is only mildly violent. We can also *copy* and *paste*. Cutting and copying add the selection to the clipboard, and pasting copies the contents of the clipboard to the current application. In Emacs, we take the homicidal approach and *kill* regions, adding them to the *kill ring*. Don't you feel *braver* and *truer* knowing that you're laying waste to untold kilobytes of text? We can then *yank*, inserting the most recently killed text at point. We can also *copy* text to the kill ring without actually killing it.

Why bother with all this morbid terminology? Well, first, so you won't be frightened when you hear someone talking about killing things in Emacs. But more important, Emacs allows you to do tasks that you can't do with the typical cut/copy/paste clipboard featureset.

Emacs stores multiple blocks of text on the kill ring, and you can cycle through them. This is cool because you can cycle through to retrieve text you killed a long time ago. Let's see this in action:

1. Create a region over the word *Treasure* in the first line.
2. Use **M-w**, which is bound to the `kill-ring-save` command. In general, **M-w** is like copying. It adds the region to the kill ring without deleting it from your buffer.
3. Move point to the word *choreography* on the last line.
4. Use **M-d**, which is bound to the `kill-word` command. This adds *choreography* to the kill ring and deletes it from your buffer.

5. Use **C-y**. This will yank the text you just killed, *choreography*, inserting it at point.
6. Use **M-y**. This will remove *choreography* and yank the next item on the kill ring, *Treasure*.

You can see some useful kill/yank key bindings in Table 2-2.

Table 2-2: Key Bindings for Killing and Yanking Text

Keys	Description
C-w	Kill region.
M-w	Copy region to kill ring.
C-y	Yank.
M-y	Cycle through kill ring after yanking.
M-d	Kill word.
C-k	Kill line.

Editing and Help

Table 2-3 shows some additional, useful, editing key bindings you should know about for dealing with spacing and expanding text.

Table 2-3: Other Useful Editing Key Bindings

Keys	Description
Tab	Indent line.
C-j	New line and indent, equivalent to ENTER followed by TAB.
M-/	Hippie expand; cycles through possible expansions of the text before point.
M-	Delete all spaces and tabs around point. (I use this one a lot.)

Emacs also has excellent built-in help. The two key bindings shown in Table 2-4 will serve you well.

Table 2-4: Key Bindings for Built-in Help

Keys	Description
C-h k *key-binding*	Describe the function bound to the key binding. To get this to work, you actually perform the key sequence after typing **C-h k**.
C-h f	Describe function.

The help text appears in a new *window*, a concept I will cover later in the chapter. For now, you can close help windows by pressing **C-x o q**.

Using Emacs with Clojure

Next, I'll explain how to use Emacs to efficiently develop a Clojure application. You'll learn how to start a REPL process that's connected to Emacs and how to work with Emacs windows. Then I'll cover a cornucopia of useful key bindings for evaluating expressions, compiling files, and performing other handy tasks. Finally, I'll show you how to handle Clojure errors and introduce some features of Paredit, an optional minor mode, which is useful for writing and editing code in Lisp-style languages.

If you want to start digging in to Clojure code, please do skip ahead! You can always return later.

Fire Up Your REPL!

As you learned in Chapter 1, a REPL allows you to interactively write and run Clojure code. The REPL is a running Clojure program that gives you a prompt and then reads your input, evaluates it, prints the result, and loops back to the prompt. In Chapter 1, you started the REPL in a terminal window with `lein repl`. In this section, you'll start a REPL directly in Emacs.

To connect Emacs to a REPL, you'll use the Emacs package CIDER, available at *https://github.com/clojure-emacs/cider/*. If you followed the configuration instructions earlier in this chapter, you should already have it installed, but you can also install it by running **M-x `package-install`**, entering **`cider`**, and pressing ENTER.

CIDER allows you to start a REPL within Emacs and provides you with key bindings that allow you to interact with the REPL more efficiently. Go ahead and start a REPL session now. Using Emacs, open the file *clojure-noob/src/clojure_noob/core.clj*, which you created in Chapter 1. Next, use **M-x `cider-jack-in`**. This starts the REPL and creates a new buffer where you can interact with it. After a short wait (it should be less than a minute), you should see something like Figure 2-8.

Figure 2-8: What your Emacs should look like after running M-x `cider-jack-in`

Now we have two windows: our *core.clj* file is open on the left, and the REPL is running on the right. If you've never seen Emacs split in half like this, don't worry! I'll talk about how Emacs splits windows in a second. In the meantime, try evaluating some code in the REPL. Type in the following bolded lines. The result that you should see printed in the REPL when you press ENTER is shown after each line of code. Don't worry about the code at this time; I'll cover all these functions in the next chapter.

```
(+ 1 2 3 4)
; => 10
(map inc [1 2 3 4])
; => (2 3 4 5)
(reduce + [5 6 100])
; => 111
```

Pretty nifty! You can use this REPL just as you used `lein repl` in the first chapter. You can also do a whole lot more, but before I go into that, I'll explain how to work with split-screen Emacs.

Interlude: Emacs Windows and Frames

Let's take a quick detour to talk about how Emacs handles frames and windows, and to go over some useful window-related key bindings. Feel free to skip this section if you're already familiar with Emacs windows.

Emacs was invented in, like, 1802 or something, so it uses terminology slightly different from what you're used to. What you would normally refer to as a *window*, Emacs calls a *frame*, and the frame can be split into multiple *windows*. Splitting into multiple windows allows you to view more than one buffer at a time. You already saw this happen when you ran `cider-jack-in` (see Figure 2-9).

Figure 2-9: In Emacs, a frame contains windows.

Table 2-5 shows several key bindings for working with the frame and windows.

Table 2-5: Emacs Window Key Bindings

Keys	Description
C-x o	Switch cursor to another window. Try this now to switch between your Clojure file and the REPL.
C-x 1	Delete all other windows, leaving only the current window in the frame. This doesn't close your buffers, and it won't cause you to lose any work.
C-x 2	Split frame above and below.
C-x 3	Split frame side by side.
C-x 0	Delete current window.

I encourage you to try the Emacs window key bindings. For example, put your cursor in the left window, the one with the Clojure file, and use **C-x 1**. The other window should disappear, and you should see only the Clojure code. Then do the following:

- Use **C-x 3** to split the window side by side again.
- Use **C-x o** to switch to the right window.
- Use **C-x b** `*cider-repl*` to switch to the CIDER buffer in the right window.

Once you've experimented a bit, set up Emacs so that it contains two side-by-side windows with Clojure code on the left and the CIDER buffer on the right, as in the previous images. If you're interested in learning more about windows and frames, the Emacs manual has a ton of info: see *http://www.gnu.org/software/emacs/manual/html_node/elisp/Windows.html#Windows*.

Now that you can navigate Emacs windows, it's time to learn some Clojure development key bindings!

A Cornucopia of Useful Key Bindings

Now you're ready to learn some key bindings that will reveal the true power of using Emacs for Clojure projects. These commands will let you evaluate, tweak, compile, and run code with just a few dainty keystrokes. Let's start by going over how to quickly evaluate an expression.

At the bottom of *core.clj*, add the following:

```
(println "Cleanliness is next to godliness")
```

Now use **C-e** to navigate to the end of the line, and then use **C-x C-e**. The text `Cleanliness is next to godliness` should appear in the CIDER buffer, as shown in Figure 2-10.

Figure 2-10: Instantly evaluating code from another buffer in the REPL

The key binding **C-x C-e** runs the command `cider-eval-last-expression`. As the name suggests, this command sends the expression immediately preceding point to the REPL, which then evaluates the expression. You can also try **C-u C-x C-e**, which prints the result of the evaluation after point.

Now let's try to run the `-main` function we wrote in Chapter 1 so we can let the world know that we're little teapots.

In the *core.clj* buffer, use **C-c M-n**. This key binding sets the namespace to the namespace listed at the top of your current file, so the prompt in the right window should now read `clojure-noob.core>`. I haven't gone into detail about namespaces yet, but for now it's enough to know that a namespace is an organizational mechanism that allows us to avoid naming conflicts. Next, enter **`(-main)`** at the prompt. The REPL should print `I'm a little teapot!` How exciting!

Now let's create a new function and run it. At the bottom of *core.clj*, add the following:

```
(defn train
  []
  (println "Choo choo!"))
```

When you're done, save your file and use **C-c C-k** to compile your current file within the REPL session. (You have to compile your code for the REPL to be aware of your changes.) Now if you run `(train)` in the REPL, it will echo back `Choo choo!`.

While still in the REPL, try **C-↑** (CTRL plus the up arrow key). **C-↑** and **C-↓** cycle through your REPL history, which includes all the Clojure expressions that you've asked the REPL to evaluate.

Note for Mac users: by default, OS X maps **C-↑**, **C-↓**, **C-←**, and **C-→** to Mission Control commands. You can change your Mac key bindings by opening System Preferences, and then going to Keyboard ▸Shortcuts ▸ Mission Control.

Finally, try this:

1. Type **`(-main`** at the REPL prompt. Note the lack of a closing parenthesis.
2. Press **C-enter**.

CIDER should close the parenthesis and evaluate the expression. This is just a nice little convenience that CIDER provides for dealing with so many parentheses.

CIDER also has a few key bindings that are great when you're learning Clojure. Pressing **C-c C-d C-d** will display documentation for the symbol under point, which can be a huge time-saver. When you're done with the documentation, press **q** to close the documentation buffer. The key binding **M-.** will navigate to the source code for the symbol under point, and **M-,** will return you to your original buffer and position. Finally, **C-c C-d C-a** lets you search for arbitrary text across function names and documentation. This is a great way to find a function when you can't exactly remember its name.

The CIDER README (*https://github.com/clojure-emacs/cider/*) has a comprehensive list of key bindings that you can learn over time, but for now, Tables 2-6 and 2-7 contain a summary of the key bindings we just went over.

Table 2-6: Clojure Buffer Key Bindings

Keys	Description
C-c M-n	Switch to namespace of current buffer.
C-x C-e	Evaluate expression immediately preceding point.
C-c C-k	Compile current buffer.
C-c C-d C-d	Display documentation for symbol under point.
M-. and M-,	Navigate to source code for symbol under point and return to your original buffer.
C-c C-d C-a	Apropros search; find arbitrary text across function names and documentation.

Table 2-7: CIDER Buffer Key Bindings

Keys	Description
C-↑, C-↓	Cycle through REPL history.
C-enter	Close parentheses and evaluate.

How to Handle Errors

In this section, you'll write some buggy code so you can see how Emacs responds to it and how you can recover from the error and continue on your merry way. You'll do this in both the REPL buffer and the *core.clj* buffer. Let's start with the REPL. At the prompt, type `(map)` and press ENTER. You should see something like Figure 2-11.

Figure 2-11: This is what happens when you run bad code in the REPL.

As you can see, calling `map` with no arguments causes Clojure to lose its mind—it shows an `ArityException` error message in your REPL buffer and fills your left window with text that looks like the ravings of a madman. These ravings are the *stack trace*, which shows the function that actually threw the exception, along with which function called *that* function, down the stack of function calls.

Clojure's stack traces can be difficult to decipher when you're just starting, but after a while you'll learn to get useful information from them. CIDER gives you a hand by allowing you to filter stack traces, which reduces noise so you can zero in on the cause of your exception. Line 2 of the `*cider-error*` buffer has the filters Clojure, Java, REPL, Tooling, Duplicates, and All. You can click each option to activate that filter. You can also click each stack trace line to jump to the corresponding source code.

Here's how to close the stack trace in the left window:

1. Use **C-x o** to switch to the window.
2. Press **q** to close the stack trace and go back to CIDER.

If you want to view the error again, you can switch to the `*cider-error*` buffer. You can also get error messages when trying to compile files. To see this, go to the *core.clj* buffer, write some buggy code, and compile:

1. Add `(map)` to the end.
2. Use **C-c C-k** to compile.

You should see a `*cider-error*` buffer similar to the one you saw earlier. Again, press **q** to close the stack trace.

Paredit

While writing code in the Clojure buffer, you may have noticed some unexpected things happening. For example, every time you type a left parenthesis, a right parenthesis immediately gets inserted.

This occurs thanks to *paredit-mode*, a minor mode that turns Lisp's profusion of parentheses from a liability into an asset. Paredit ensures that all parentheses, double quotes, and brackets are closed, relieving you of that odious burden.

Paredit also offers key bindings to easily navigate and alter the structure created by all those parentheses. In the next section, I'll go over the most useful key bindings, but you can also check out a comprehensive cheat sheet at *https://github.com/georgek/paredit-cheatsheet/blob/master/paredit-cheatsheet.pdf* (in the cheat sheet, the red pipe represents point).

However, if you're not used to it, paredit can sometimes be annoying. I think it's more than worth your while to take some time to learn it, but you can always disable it with **M-x `paredit-mode`**, which toggles the mode on and off.

The following section shows you the most useful key bindings.

Wrapping and Slurping

Wrapping surrounds the expression after point with parentheses. *Slurping* moves a closing parenthesis to include the next expression to the right. For example, say we start with this:

```
(+ 1 2 3 4)
```

and we want to get this:

```
(+ 1 (* 2 3) 4)
```

We can wrap the `2`, add an asterisk, and then slurp the `3`. First, place point, which is represented here as a vertical pipe, `|`:

```
(+ 1 |2 3 4)
```

Then type **M-(**, the binding for *paredit-wrap-round*, getting this result:

```
(+ 1 (|2) 3 4)
```

Add the asterisk and a space:

```
(+ 1 (* |2) 3 4)
```

To slurp in the `3`, press **C-→**:

```
(+ 1 (* |2 3) 4)
```

This makes it easy to add and extend parentheses without wasting precious moments holding down arrow keys to move point.

Barfing

Suppose, in the preceding example, you accidentally slurped the four. To unslurp it (also known as *barfing*), place your cursor (`|`) anywhere in the inner parentheses:

```
(+ 1 (|* 2 3 4))
```

Then use **C-←**:

```
(+ 1 (|* 2 3) 4)
```

Ta-da! Now you know how to expand and contract parentheses at will.

Navigation

Often when writing in a Lisp dialect, you'll work with expressions like this:

```
(map (comp record first)
     (d/q '[:find ?post
            :in $ ?search
            :where
            [(fulltext $ :post/content ?search)
             [[?post ?content]]]]
          (db/db)
          (:q params)))
```

With this kind of expression, it's useful to jump quickly from one subexpression to the next. If you put point right before an opening parenthesis, **C-M-f** will take you to the closing parenthesis. Similarly, if point is right after a closing parenthesis, **C-M-b** will take you to the opening parenthesis.

Table 2-8 summarizes the paredit key bindings you just learned.

Table 2-8: Paredit Key Bindings

Keys	Description
M-x `paredit-mode`	Toggle paredit mode.
M-(	Surround expression after point in parentheses (paredit-wrap-round).
C-→	Slurp; move closing parenthesis to the right to include next expression.
C-←	Barf; move closing parenthesis to the left to exclude last expression.
C-M-f, **C-M-b**	Move to the opening/closing parenthesis.

Continue Learning

Emacs is one of the longest-lived editors, and its adherents often approach fanaticism in their enthusiasm for it. It can be awkward to use at first, but stick with it and you will be amply rewarded over your lifetime.

Whenever I open Emacs, I feel inspired. Like a craftsman entering his workshop, I feel a realm of possibility open before me. I feel the comfort of an environment that has evolved over time to fit me perfectly—an assortment of packages and key bindings that help me bring ideas to life day after day.

These resources will help you as you continue on your Emacs journey:

- *The Emacs Manual* provides excellent, comprehensive instructions. Spend some time with it every morning! Download the PDF and read it on the go: *http://www.gnu.org/software/emacs/manual/html_node/emacs/index.html#Top*.
- *The Emacs Reference Card* is a handy cheat sheet: *http://www.ic.unicamp.br/~helio/disciplinas/MC102/Emacs_Reference_Card.pdf*.
- *Mastering Emacs* by Mickey Petersen is one of the best Emacs resources available. Start with the reading guide: *http://www.masteringemacs.org/reading-guide/*.
- For the more visually minded folks, I recommend the hand-drawn "How to Learn Emacs: A Beginner's Guide to Emacs 24 or Later" by Sacha Chua: *http://sachachua.com/blog/wp-content/uploads/2013/05/How-to-Learn-Emacs8.png*.
- Just press **C-h t** for the built-in tutorial.

Summary

Whew! You've covered a lot of ground. You now know about Emacs's true nature as a Lisp interpreter. Key bindings act as shortcuts to execute elisp functions, and modes are collections of key bindings and functions. You learned how to interact with Emacs on its own terms and mastered buffers, windows, regions, killing, and yanking. Finally, you learned how to easily work with Clojure using CIDER and paredit.

With all of this hard-won Emacs knowledge under your belt, it's time to start learning Clojure in earnest!

PART II

LANGUAGE FUNDAMENTALS

3

DO THINGS: A CLOJURE CRASH COURSE

It's time to learn how to actually *do things* with Clojure! Hot damn! Although you've undoubtedly heard of Clojure's awesome concurrency support and other stupendous features, Clojure's most salient characteristic is that it is a Lisp. In this chapter, you'll explore the elements that compose this Lisp core: syntax, functions, and data. Together they will give you a solid foundation for representing and solving problems in Clojure.

After laying this groundwork, you will be able to write some super important code. In the last section, you'll tie everything together by creating a model of a hobbit and writing a function to hit it in a random spot. Super! Important!

As you move through the chapter, I recommend that you type the examples in a REPL and run them. Programming in a new language is a skill, and just like yodeling or synchronized swimming, you have to practice to learn it. By the way, *Synchronized Swimming for Yodelers for the Brave and True* will be published in August of 20never. Keep an eye out for it!

Syntax

Clojure's syntax is simple. Like all Lisps, it employs a uniform structure, a handful of special operators, and a constant supply of parentheses delivered from the parenthesis mines hidden beneath the Massachusetts Institute of Technology, where Lisp was born.

Forms

All Clojure code is written in a uniform structure. Clojure recognizes two kinds of structures:

- Literal representations of data structures (like numbers, strings, maps, and vectors)
- Operations

We use the term *form* to refer to valid code. I'll also sometimes use *expression* to refer to Clojure forms. But don't get too hung up on the terminology. Clojure *evaluates* every form to produce a value. These literal representations are all valid forms:

```
1
"a string"
["a" "vector" "of" "strings"]
```

Your code will rarely contain free-floating literals, of course, because they don't actually do anything on their own. Instead, you'll use literals in operations. Operations are how you *do things*. All operations take the form *opening parenthesis, operator, operands, closing parenthesis*:

```
(operator operand1 operand2 ... operandn)
```

Notice that there are no commas. Clojure uses whitespace to separate operands, and it treats commas as whitespace. Here are some example operations:

```
(+ 1 2 3)
; => 6

(str "It was the panda " "in the library " "with a dust buster")
; => "It was the panda in the library with a dust buster"
```

In the first operation, the operator `+` adds the operands `1`, `2`, and `3`. In the second operation, the operator `str` concatenates three strings to form a new string. Both are valid forms. Here's something that is not a form because it doesn't have a closing parenthesis:

```
(+
```

Clojure's structural uniformity is probably different from what you're used to. In other languages, different operations might have different structures depending on the operator and the operands. For example, JavaScript employs a smorgasbord of infix notation, dot operators, and parentheses:

```
1 + 2 + 3
"It was the panda ".concat("in the library ", "with a dust buster")
```

Clojure's structure is very simple and consistent by comparison. No matter which operator you're using or what kind of data you're operating on, the structure is the same.

Control Flow

Let's look at three basic control flow operators: `if`, `do`, and `when`. Throughout the book you'll encounter more, but these will get you started.

if

This is the general structure for an `if` expression:

```
(if boolean-form
  then-form
  optional-else-form)
```

A Boolean form is just a form that evaluates to a truthy or falsey value. You'll learn about truthiness and falsiness in the next section. Here are a couple of `if` examples:

```
(if true
  "By Zeus's hammer!"
  "By Aquaman's trident!")
; => "By Zeus's hammer!"

(if false
  "By Zeus's hammer!"
  "By Aquaman's trident!")
; => "By Aquaman's trident!"
```

The first example returns `"By Zeus's hammer!"` because the Boolean form evaluates to `true`, a truthy value, and the second example returns `"By Aquaman's trident!"` because its Boolean form, `false`, evaluates to a falsey value.

You can also omit the `else` branch. If you do that and the Boolean expression is false, Clojure returns `nil`, like this:

```
(if false
  "By Odin's Elbow!")
; => nil
```

Notice that `if` uses operand position to associate operands with the `then` and `else` branches: the first operand is the `then` branch, and the second operand is the (optional) `else` branch. As a result, each branch can have only one form. This is different from most languages. For example, you can write this in Ruby:

```
if true
  doer.do_thing(1)
  doer.do_thing(2)
else
  other_doer.do_thing(1)
  other_doer.do_thing(2)
end
```

To get around this apparent limitation, you can use the `do` operator.

do

The `do` operator lets you *wrap up* multiple forms in parentheses and run each of them. Try the following in your REPL:

```
(if true
  (do (println "Success!")
      "By Zeus's hammer!")
  (do (println "Failure!")
      "By Aquaman's trident!"))
; => Success!
; => "By Zeus's hammer!"
```

; tried using `do` to print 2 different lines, didn't work b/c the second line is returned as the value of the if expression (i wasn't using a conditional)

This operator lets you do multiple things in each of the `if` expression's branches. In this case, two things happen: `Success!` is printed in the REPL, and `"By Zeus's hammer!"` is returned as the value of the entire `if` expression.

when

The `when` operator is like a combination of `if` and `do`, but with no `else` branch. Here's an example:

```
(when true
  (println "Success!")
  "abra cadabra")
```

```
; => Success!
; => "abra cadabra"
```

;=> Use `when` if you want to do multiple things when some condition is true, and you always want to return `nil` when the condition is false.

nil, true, false, Truthiness, Equality, and Boolean Expressions

Clojure has `true` and `false` values. `nil` is used to indicate *no value* in Clojure. You can check if a value is `nil` with the appropriately named `nil?` function:

```
(nil? 1)
; => false

(nil? nil)
; => true
```

Both `nil` and `false` are used to represent logical falsiness, whereas all other values are logically truthy. *Truthy* and *falsey* refer to how a value is treated in a Boolean expression, like the first expression passed to `if`:

```
(if "bears eat beets"
  "bears beets Battlestar Galactica")
; => "bears beets Battlestar Galactica"

(if nil
  "This won't be the result because nil is falsey"
  "nil is falsey")
; => "nil is falsey"
```

In the first example, the string `"bears eat beets"` is considered truthy, so the `if` expression evaluates to `"bears beets Battlestar Galactica"`. The second example shows a falsey value as falsey.

Clojure's equality operator is `=`:

```
(= 1 1)
; => true

(= nil nil)
; => true

(= 1 2)
; => false
```

Some other languages require you to use different operators when comparing values of different types. For example, you might have to use some kind of special string equality operator made just for strings. But you don't need anything weird or tedious like that to test for equality when using Clojure's built-in data structures.

Clojure uses the Boolean operators `or` and `and`. `or` returns either the first truthy value or the last value. `and` returns the first falsey value or, if no values are falsey, the last truthy value. Let's look at `or` first:

```
(or false nil :large_I_mean_venti :why_cant_I_just_say_large)
; => :large_I_mean_venti

(or (= 0 1) (= "yes" "no"))
; => false

(or nil)
; => nil
```

In the first example, the return value is `:large_I_mean_venti` because it's the first truthy value. The second example has no truthy values, so `or` returns the last value, which is `false`. In the last example, once again no truthy values exist, and `or` returns the last value, which is `nil`. Now let's look at `and`:

```
(and :free_wifi :hot_coffee)
; => :hot_coffee

(and :feelin_super_cool nil false)
; => nil
```

In the first example, `and` returns the last truthy value, `:hot_coffee`. In the second example, `and` returns `nil`, which is the first falsey value.

Naming Values with def

You use `def` to *bind* a name to a value in Clojure:

```
(def failed-protagonist-names
  ["Larry Potter" "Doreen the Explorer" "The Incredible Bulk"])

failed-protagonist-names
; => ["Larry Potter" "Doreen the Explorer" "The Incredible Bulk"]
```

In this case, you're binding the name `failed-protagonist-names` to a vector containing three strings (you'll learn about vectors in "Vectors" on page 45).

Notice that I'm using the term *bind*, whereas in other languages you'd say you're *assigning* a value to a *variable*. Those other languages typically encourage you to perform multiple assignments to the same variable.

For example, in Ruby you might perform multiple assignments to a variable to build up its value:

```
severity = :mild
error_message = "OH GOD! IT'S A DISASTER! WE'RE "
if severity == :mild
  error_message = error_message + "MILDLY INCONVENIENCED!"
else
  error_message = error_message + "DOOOOOOOMED!"
end
```

You might be tempted to do something similar in Clojure:

```
(def severity :mild)
(def error-message "OH GOD! IT'S A DISASTER! WE'RE ")
(if (= severity :mild)
  (def error-message (str error-message "MILDLY INCONVENIENCED!"))
  (def error-message (str error-message "DOOOOOOOMED!")))
```

However, changing the value associated with a name like this can make it harder to understand your program's behavior because it's more difficult to know which value is associated with a name or why that value might have changed. Clojure has a set of tools for dealing with change, which you'll learn about in Chapter 10. As you learn Clojure, you'll find that you'll rarely need to alter a name/value association. Here's one way you could write the preceding code:

```
(defn error-message
  [severity]
  (str "OH GOD! IT'S A DISASTER! WE'RE "
       (if (= severity :mild)
         "MILDLY INCONVENIENCED!"     ; this reads really well!
         "DOOOOOOOMED!")))

(error-message :mild)
; => "OH GOD! IT'S A DISASTER! WE'RE MILDLY INCONVENIENCED!"
```

Here, you create a function, `error-message`, which accepts a single argument, `severity`, and uses that to determine which string to return. You then call the function with `:mild` for the severity. You'll learn all about creating functions in "Functions" on page 48; in the meantime, you should treat `def` as if it's defining constants. In the next few chapters, you'll learn how to work with this apparent limitation by embracing the functional programming paradigm.

Data Structures

Clojure comes with a handful of data structures that you'll use the majority of the time. If you're coming from an object-oriented background, you'll be surprised at how much you can do with the seemingly basic types presented here.

All of Clojure's data structures are immutable, meaning you can't change them in place. For example, in Ruby you could do the following to reassign the failed protagonist name at index 0:

```
failed_protagonist_names = [
  "Larry Potter",
  "Doreen the Explorer",
  "The Incredible Bulk"
]
failed_protagonist_names[0] = "Gary Potter"

failed_protagonist_names
# => [
#   "Gary Potter",
#   "Doreen the Explorer",
#   "The Incredible Bulk"
# ]
```

Clojure has no equivalent for this. You'll learn more about why Clojure was implemented this way in Chapter 10, but for now it's fun to learn just how to do things without all that philosophizing. Without further ado, let's look at numbers in Clojure.

Numbers

Clojure has pretty sophisticated numerical support. I won't spend much time dwelling on the boring technical details (like coercion and contagion), because that will get in the way of *doing things*. If you're interested in said boring details, check out the documentation at *http://clojure.org/data_structures#Data%20Structures-Numbers*. Suffice it to say, Clojure will merrily handle pretty much anything you throw at it. ; visited 3/20/17, revisit

In the meantime, we'll work with integers and floats. We'll also work with ratios, which Clojure can represent directly. Here's an integer, a float, and a ratio, respectively:

```
93
1.2
1/5
```

Strings

Strings represent text. The name comes from the ancient Phoenicians, who one day invented the alphabet after an accident involving yarn. Here are some examples of string literals: ; ahahaha

```
"Lord Voldemort"
"\"He who must not be named\""
"\"Great cow of Moscow!\" - Hermes Conrad"
```

Notice that Clojure only allows double quotes to delineate strings. `'Lord Voldemort'`, for example, is not a valid string. Also notice that Clojure doesn't have string interpolation. It only allows concatenation via the `str` function:

```
(def name "Chewbacca")
(str "\"Uggllglglglglglglgllll\" - " name)
; => "Uggllglglglglglglgllll" - Chewbacca
```

Maps

Maps are similar to dictionaries or hashes in other languages. They're a way of associating some value with some other value. The two kinds of maps in Clojure are hash maps and sorted maps. I'll only cover the more basic hash maps. Let's look at some examples of map literals. Here's an empty map:

```
{}
```

In this example, `:first-name` and `:last-name` are keywords (I'll cover those in the next section):

```
{:first-name "Charlie"
 :last-name "McFishwich"}
```

Here we associate `"string-key"` with the `+` function:

```
{"string-key" +}
```

Maps can be nested:

```
{:name {:first "John" :middle "Jacob" :last "Jingleheimerschmidt"}}
```

Notice that map values can be of any type—strings, numbers, maps, vectors, even functions. Clojure don't care!

Besides using map literals, you can use the `hash-map` function to create a map:

```
(hash-map :a 1 :b 2)
; => {:a 1 :b 2}
```

You can look up values in maps with the `get` function:

```
(get {:a 0 :b 1} :b)
; => 1

(get {:a 0 :b {:c "ho hum"}} :b)
; => {:c "ho hum"}
```

In both of these examples, we asked `get` for the value of the `:b` key in the given map—in the first case it returns `1`, and in the second case it returns the nested map `{:c "ho hum"}`.

`get` will return `nil` if it doesn't find your key, or you can give it a default value to return, such as `"unicorns?"`:

```
(get {:a 0 :b 1} :c)
; => nil

(get {:a 0 :b 1} :c "unicorns?")
; => "unicorns?"      ; ^^ default value
```

The `get-in` function lets you look up values in nested maps:

```
(get-in {:a 0 :b {:c "ho hum"}} [:b :c])   ; so you pass it an array-like thing mapping
; => "ho hum"                                to the nested value
```

Another way to look up a value in a map is to treat the map like a function with the key as its argument: ; cool

```
({:name "The Human Coffeepot"} :name)
; => "The Human Coffeepot"
```

Another cool thing you can do with maps is use keywords as functions to look up their values, which leads to the next subject, keywords.

Keywords

Clojure keywords are best understood by seeing how they're used. They're primarily used as keys in maps, as you saw in the preceding section. Here are some more examples of keywords:

```
:a
:rumplestiltsken
:34
:_?
```

Keywords can be used as functions that look up the corresponding value in a data structure. For example, you can look up `:a` in a map:

```
(:a {:a 1 :b 2 :c 3})
; => 1
```

; so intuitive, wow

This is equivalent to:

```
(get {:a 1 :b 2 :c 3} :a)
; => 1
```

You can provide a default value, as with `get`:

```
(:d {:a 1 :b 2 :c 3} "No gnome knows homes like Noah knows")
; => "No gnome knows homes like Noah knows"
```

Using a keyword as a function is pleasantly succinct, and Real Clojurists do it all the time. You should do it too!

Vectors

A vector is similar to an array, in that it's a 0-indexed collection. For example, here's a vector literal:

```
[3 2 1]
```

Here we're returning the 0th element of a vector:

```
(get [3 2 1] 0)
; => 3
```

Here's another example of getting by index:

```
(get ["a" {:name "Pugsley Winterbottom"} "c"] 1)
; => {:name "Pugsley Winterbottom"}
```

You can see that vector elements can be of any type, and you can mix types. Also notice that we're using the same `get` function as we use when looking up values in maps.

You can create vectors with the `vector` function:

```
(vector "creepy" "full" "moon")
; => ["creepy" "full" "moon"]
```

You can use the `conj` function to add additional elements to the vector. Elements are added to the *end* of a vector:

```
(conj [1 2 3] 4)
; => [1 2 3 4]
```

Vectors aren't the only way to store sequences; Clojure also has *lists*.

Lists

Lists are similar to vectors in that they're linear collections of values. But there are some differences. For example, you can't retrieve list elements with `get`. To write a list literal, just insert the elements into parentheses and use a single quote at the beginning:

```
'(1 2 3 4)
; => (1 2 3 4)
```

Notice that when the REPL prints out the list, it doesn't include the single quote. We'll come back to why that happens later, in Chapter 7. If you want to retrieve an element from a list, you can use the `nth` function:

```
(nth '(:a :b :c) 0)
; => :a

(nth '(:a :b :c) 2)
; => :c
```

I don't cover performance in detail in this book because I don't think it's useful to focus on it until after you've become familiar with a language. However, it's good to know that using `nth` to retrieve an element from a list is slower than using `get` to retrieve an element from a vector. This is because Clojure has to traverse all *n* elements of a list to get to the *n*th, whereas it only takes a few hops at most to access a vector element by its index.

List values can have any type, and you can create lists with the `list` function:

```
(list 1 "two" {3 4})
; => (1 "two" {3 4})
```

Elements are added to the *beginning* of a list:

```
(conj '(1 2 3) 4)
; => (4 1 2 3)
```

When should you use a list and when should you use a vector? A good rule of thumb is that if you need to easily add items to the beginning of a sequence or if you're writing a macro, you should use a list. Otherwise, you should use a vector. As you learn more, you'll get a good feel for when to use which.

Sets

Sets are collections of unique values. Clojure has two kinds of sets: hash sets and sorted sets. I'll focus on hash sets because they're used more often. Here's the literal notation for a hash set:

```
#{"kurt vonnegut" 20 :icicle}
```

You can also use `hash-set` to create a set:

```
(hash-set 1 1 2 2)
; => #{1 2}
```

; so it just filters out duplicate values

Note that multiple instances of a value become one unique value in the set, so we're left with a single `1` and a single `2`. If you try to add a value

to a set that already contains that value (such as `:b` in the following code), it will still have only one of that value:

```
(conj #{:a :b} :b)
; => #{:a :b}
```

You can also create sets from existing vectors and lists by using the `set` function:

```
(set [3 3 3 4 4])
; => #{3 4}
```

You can check for set membership using the `contains?` function, by using `get`, or by using a keyword as a function with the set as its argument. `contains?` returns `true` or `false`, whereas `get` and keyword lookup will return the value if it exists, or `nil` if it doesn't.

Here's how you'd use `contains?`:

```
(contains? #{:a :b} :a)
; => true

(contains? #{:a :b} 3)
; => false

(contains? #{nil} nil)
; => true
```

Here's how you'd use a keyword:

```
(:a #{:a :b})
; => :a
```

And here's how you'd use `get`:

```
(get #{:a :b} :a)
; => :a

(get #{:a nil} nil)
; => nil

(get #{:a :b} "kurt vonnegut")
; => nil
```

Notice that using `get` to test whether a set contains `nil` will always return `nil`, which is confusing. `contains?` may be the better option when you're testing specifically for set membership.

Simplicity

You may have noticed that the treatment of data structures so far doesn't include a description of how to create new types or classes. The reason is that Clojure's emphasis on simplicity encourages you to reach for the built-in data structures first.

If you come from an object-oriented background, you might think that this approach is weird and backward. However, what you'll find is that your data does not have to be tightly bundled with a class for it to be useful and intelligible. Here's an epigram loved by Clojurists that hints at the Clojure philosophy:

> It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures.
> —Alan Perlis

You'll learn more about this aspect of Clojure's philosophy in the coming chapters. For now, keep an eye out for the ways that you gain code reusability by sticking to basic data structures.

This concludes our Clojure data structures primer. Now it's time to dig in to functions and learn how to use these data structures!

Functions

One of the reasons people go nuts over Lisps is that these languages let you build programs that behave in complex ways, yet the primary building block—the function—is so simple. This section initiates you into the beauty and elegance of Lisp functions by explaining the following:

- Calling functions
- How functions differ from macros and special forms
- Defining functions
- Anonymous functions
- Returning functions

Calling Functions

By now you've seen many examples of function calls:

```
(+ 1 2 3 4)
(* 1 2 3 4)
(first [1 2 3 4])
```

Remember that all Clojure operations have the same syntax: opening parenthesis, operator, operands, closing parenthesis. *Function call* is just another term for an operation where the operator is a function or a *function expression* (an expression that returns a function).

This lets you write some pretty interesting code. Here's a function expression that returns the + (addition) function:

```
(or + -)
; => #<core$_PLUS_ clojure.core$_PLUS_@76dace31>
```

That return value is the string representation of the addition function. Because the return value of `or` is the first truthy value, and here the addition function is truthy, the addition function is returned. You can also use this expression as the operator in another expression:

```
((or + -) 1 2 3)
; => 6
```

Because `(or + -)` returns `+`, this expression evaluates to the sum of `1`, `2`, and `3`, returning `6`.

Here are a couple more valid function calls that each return `6`:

```
((and (= 1 1) +) 1 2 3)
; => 6

((first [+ 0]) 1 2 3)
; => 6
```

In the first example, the return value of `and` is the first falsey value or the last truthy value. In this case, `+` is returned because it's the last truthy value, and is then applied to the arguments `1 2 3`, returning `6`. In the second example, the return value of `first` is the first element in a sequence, which is `+` in this case.

However, these aren't valid function calls, because numbers and strings aren't functions:

```
(1 2 3 4)
("test" 1 2 3)
```

If you run these in your REPL, you'll get something like this:

```
ClassCastException java.lang.String cannot be cast to clojure.lang.IFn
user/eval728 (NO_SOURCE_FILE:1)
```

You're likely to see this error many times as you continue with Clojure: *<x> cannot be cast to clojure.lang.IFn* just means that you're trying to use something as a function when it's not.

Function flexibility doesn't end with the function expression! Syntactically, functions can take any expressions as arguments—including *other functions*. Functions that can either take a function as an argument or return a function are called *higher-order functions*. Programming languages with higher-order functions are said to support *first-class functions* because you can treat functions as values in the same way you treat more familiar data types like numbers and vectors.

Take the `map` function (not to be confused with the map data structure), for instance. `map` creates a new list by applying a function to each member of a collection. Here, the `inc` function increments a number by 1:

```
(inc 1.1)
; => 2.1

(map inc [0 1 2 3])
; => (1 2 3 4)
```

(Note that `map` doesn't return a vector, even though we supplied a vector as an argument. You'll learn why in Chapter 4. For now, just trust that this is okay and expected.)

Clojure's support for first-class functions allows you to build more powerful abstractions than you can in languages without them. Those unfamiliar with this kind of programming think of functions as allowing you to generalize operations over data instances. For example, the `+` function abstracts addition over any specific numbers.

By contrast, Clojure (and all Lisps) allows you to create functions that generalize over processes. `map` allows you to generalize the process of transforming a collection by applying a function—any function—over any collection.

The last detail that you need know about function calls is that Clojure evaluates all function arguments recursively before passing them to the function. Here's how Clojure would evaluate a function call whose arguments are also function calls:

```
(+ (inc 199) (/ 100 (- 7 2)))
(+ 200 (/ 100 (- 7 2))) ; evaluated "(inc 199)"
(+ 200 (/ 100 5)) ; evaluated (- 7 2)
(+ 200 20) ; evaluated (/ 100 5)
220 ; final evaluation
```

The function call kicks off the evaluation process, and all subforms are evaluated before applying the `+` function.

Function Calls, Macro Calls, and Special Forms

In the previous section, you learned that function calls are expressions that have a function expression as the operator. The two other kinds of expressions are *macro calls* and *special forms*. You've already seen a couple of special forms: definitions and `if` expressions.

You'll learn everything there is to know about macro calls and special forms in Chapter 7. For now, the main feature that makes special forms "special" is that, unlike function calls, *they don't always evaluate all of their operands*.

Take `if`, for example. This is its general structure:

```
(if boolean-form
  then-form
  optional-else-form)
```

Now imagine you had an `if` statement like this:

```
(if good-mood
  (tweet walking-on-sunshine-lyrics)
  (tweet mopey-country-song-lyrics))
```

Clearly, in an `if` expression like this, we want Clojure to evaluate only one of the two branches. If Clojure evaluated both `tweet` function calls, your Twitter followers would end up very confused.

Another feature that differentiates special forms is that you can't use them as arguments to functions. In general, special forms implement core Clojure functionality that just can't be implemented with functions. Clojure has only a handful of special forms, and it's pretty amazing that such a rich language is implemented with such a small set of building blocks.

Macros are similar to special forms in that they evaluate their operands differently from function calls, and they also can't be passed as arguments to functions. But this detour has taken long enough; it's time to learn how to define functions!

Defining Functions

Function definitions are composed of five main parts:

- `defn`
- Function name
- A docstring describing the function (optional)
- Parameters listed in brackets
- Function body

Here's an example of a function definition and a sample call of the function:

```
❶ (defn too-enthusiastic
❷   "Return a cheer that might be a bit too enthusiastic"
❸   [name]
❹   (str "OH. MY. GOD! " name " YOU ARE MOST DEFINITELY LIKE THE BEST "
    "MAN SLASH WOMAN EVER I LOVE YOU AND WE SHOULD RUN AWAY SOMEWHERE"))

(too-enthusiastic "Zelda")
; => "OH. MY. GOD! Zelda YOU ARE MOST DEFINITELY LIKE THE BEST MAN SLASH WOMAN
EVER I LOVE YOU AND WE SHOULD RUN AWAY SOMEWHERE"
```

At ❶, `too-enthusiastic` is the name of the function, and it's followed by a descriptive docstring at ❷. The parameter, `name`, is given at ❸, and the function body at ❹ takes the parameter and does what it says on the tin—returns a cheer that might be a bit too enthusiastic.

Let's dive deeper into the docstring, parameters, and function body.

The Docstring

The *docstring* is a useful way to describe and document your code. You can view the docstring for a function in the REPL with `(doc fn-name)`—for example, `(doc map)`. The docstring also comes into play if you use a tool to generate documentation for your code.

Parameters and Arity

Clojure functions can be defined with zero or more parameters. The values you pass to functions are called *arguments*, and the arguments can be of any type. The number of parameters is the function's *arity*. Here are some function definitions with different arities:

```
(defn no-params
  []
  "I take no parameters!")
(defn one-param
  [x]
  (str "I take one parameter: " x))
(defn two-params
  [x y]
  (str "Two parameters! That's nothing! Pah! I will smoosh them "
  "together to spite you! " x y))
```

In these examples, `no-params` is a 0-arity function, `one-param` is 1-arity, and `two-params` is 2-arity.

Functions also support *arity overloading*. This means that you can define a function so a different function body will run depending on the arity. Here's the general form of a multiple-arity function definition. Notice that each arity definition is enclosed in parentheses and has an argument list:

```
(defn multi-arity
  ;; 3-arity arguments and body
  ([first-arg second-arg third-arg]
     (do-things first-arg second-arg third-arg))
  ;; 2-arity arguments and body
  ([first-arg second-arg]
     (do-things first-arg second-arg))
  ;; 1-arity arguments and body
  ([first-arg]
     (do-things first-arg)))
```

Arity overloading is one way to provide default values for arguments. In the following example, `"karate"` is the default argument for the `chop-type` parameter:

```
(defn x-chop
  "Describe the kind of chop you're inflicting on someone"
  ([name chop-type]
     (str "I " chop-type " chop " name "! Take that!"))
  ([name]
     (x-chop name "karate")))
```

If you call `x-chop` with two arguments, the function works just as it would if it weren't a multiple-arity function:

```
(x-chop "Kanye West" "slap")
; => "I slap chop Kanye West! Take that!"
```

If you call `x-chop` with only one argument, `x-chop` will actually call itself with the second argument `"karate"` supplied:

```
(x-chop "Kanye East")
; => "I karate chop Kanye East! Take that!"
```

It might seem unusual to define a function in terms of itself like this. If so, great! You're learning a new way to do things!

You can also make each arity do something completely unrelated:

```
(defn weird-arity
  ([]
     "Destiny dressed you this morning, my friend, and now Fear is
     trying to pull off your pants. If you give up, if you give in,
     you're gonna end up naked with Fear just standing there laughing
     at your dangling unmentionables! - the Tick")
  ([number]
     (inc number)))
```

The 0-arity body returns a wise quote, and the 1-arity body increments a number. Most likely, you wouldn't want to write a function like this, because it would be confusing to have two function bodies that are completely unrelated.

Clojure also allows you to define variable-arity functions by including a *rest parameter*, as in "put the rest of these arguments in a list with the following name." The rest parameter is indicated by an ampersand (&), as shown at ❶:

```
(defn codger-communication
  [whippersnapper]
  (str "Get off my lawn, " whippersnapper "!!!"))

(defn codger
❶ [& whippersnappers]
  (map codger-communication whippersnappers))

(codger "Billy" "Anne-Marie" "The Incredible Bulk")
; => ("Get off my lawn, Billy!!!"
      "Get off my lawn, Anne-Marie!!!"
      "Get off my lawn, The Incredible Bulk!!!")
```

As you can see, when you provide arguments to variable-arity functions, the arguments are treated as a list. You can mix rest parameters with normal parameters, but the rest parameter has to come last:

```
(defn favorite-things
  [name & things]
  (str "Hi, " name ", here are my favorite things: "
       (clojure.string/join ", " things)))

(favorite-things "Doreen" "gum" "shoes" "kara-te")
; => "Hi, Doreen, here are my favorite things: gum, shoes, kara-te"
```

Finally, Clojure has a more sophisticated way of defining parameters, called *destructuring*, which deserves its own subsection.

Destructuring

The basic idea behind destructuring is that it lets you concisely bind names to values within a collection. Let's look at a basic example:

```
;; Return the first element of a collection
(defn my-first
  [[first-thing]] ; Notice that first-thing is within a vector
  first-thing)

(my-first ["oven" "bike" "war-axe"])
; => "oven"
```

Here, the `my-first` function associates the symbol `first-thing` with the first element of the vector that was passed in as an argument. You tell `my-first` to do this by placing the symbol `first-thing` within a vector.

That vector is like a huge sign held up to Clojure that says, "Hey! This function is going to receive a list or a vector as an argument. Make my life easier by taking apart the argument's structure for me and associating

meaningful names with different parts of the argument!" When destructuring a vector or list, you can name as many elements as you want and also use rest parameters:

```
(defn chooser
  [[first-choice second-choice & unimportant-choices]]
  (println (str "Your first choice is: " first-choice))
  (println (str "Your second choice is: " second-choice))
  (println (str "We're ignoring the rest of your choices. "
                "Here they are in case you need to cry over them: "
                (clojure.string/join ", " unimportant-choices))))

(chooser ["Marmalade", "Handsome Jack", "Pigpen", "Aquaman"])
; => Your first choice is: Marmalade
; => Your second choice is: Handsome Jack
; => We're ignoring the rest of your choices. Here they are in case \
     you need to cry over them: Pigpen, Aquaman
```

Here, the rest parameter `unimportant-choices` handles any number of additional choices from the user after the first and second.

You can also destructure maps. In the same way that you tell Clojure to destructure a vector or list by providing a vector as a parameter, you destructure maps by providing a map as a parameter:

```
(defn announce-treasure-location
❶ [{lat :lat lng :lng}]
  (println (str "Treasure lat: " lat))
  (println (str "Treasure lng: " lng)))

(announce-treasure-location {:lat 28.22 :lng 81.33})
; => Treasure lat: 100
; => Treasure lng: 50
```

Let's look at the line at ❶ in more detail. This is like telling Clojure, "Yo! Clojure! Do me a flava and associate the name `lat` with the value corresponding to the key `:lat`. Do the same thing with `lng` and `:lng`, okay?"

We often want to just break keywords out of a map, so there's a shorter syntax for that. This has the same result as the previous example:

; so…is :keys a reserved phrase, or is it just a placeholder and can be named anything?

```
(defn announce-treasure-location
  [{:keys [lat lng]}]
  (println (str "Treasure lat: " lat))
  (println (str "Treasure lng: " lng)))
```

You can retain access to the original map argument by using the `:as` keyword. In the following example, the original map is accessed with `treasure-location`:

```
(defn receive-treasure-location
  [{:keys [lat lng] :as treasure-location}]
```

; whoa, this is pretty weird

```
  (println (str "Treasure lat: " lat))
  (println (str "Treasure lng: " lng))

  ;; One would assume that this would put in new coordinates for your ship
  (steer-ship! treasure-location))
```

In general, you can think of destructuring as instructing Clojure on how to associate names with values in a list, map, set, or vector. Now, on to the part of the function that actually does something: the function body!

Function Body

The function body can contain forms of any kind. Clojure automatically returns the last form evaluated. This function body contains just three forms, and when you call the function, it spits out the last form, `"joe"`:

```
(defn illustrative-function
  []
  (+ 1 304)
  30
  "joe")

(illustrative-function)
; => "joe"
```

Here's another function body, which uses an `if` expression:

```
(defn number-comment
  [x]
  (if (> x 6)
    "Oh my gosh! What a big number!"
    "That number's OK, I guess"))

(number-comment 5)
; => "That number's OK, I guess"

(number-comment 7)
; => "Oh my gosh! What a big number!"
```

All Functions Are Created Equal

One final note: Clojure has no privileged functions. `+` is just a function, `-` is just a function, and `inc` and `map` are just functions. They're no better than the functions you define yourself. So don't let them give you any lip!

More important, this fact helps demonstrate Clojure's underlying simplicity. In a way, Clojure is very dumb. When you make a function call, Clojure just says, "`map`? Sure, whatever! I'll just apply this and move on." It doesn't care what the function is or where it came from; it treats all functions the same. At its core, Clojure doesn't give two burger flips about addition, multiplication, or mapping. It just cares about applying functions.

As you continue to program with Clojure, you'll see that this simplicity is ideal. You don't have to worry about special rules or syntax for working with different functions. They all work the same!

Anonymous Functions

In Clojure, functions don't need to have names. In fact, you'll use *anonymous* functions all the time. How mysterious! You create anonymous functions in two ways. The first is to use the `fn` form:

```
(fn [param-list]
  function body)
```

Looks a lot like `defn`, doesn't it? Let's try a couple of examples:

```
(map (fn [name] (str "Hi, " name))
     ["Darth Vader" "Mr. Magoo"])
; => ("Hi, Darth Vader" "Hi, Mr. Magoo")

((fn [x] (* x 3)) 8)
; => 24
```

You can treat `fn` nearly identically to the way you treat `defn`. The parameter lists and function bodies work exactly the same. You can use argument destructuring, rest parameters, and so on. You could even associate your anonymous function with a name, which shouldn't come as a surprise (if that does come as a surprise, then . . . Surprise!):

```
(def my-special-multiplier (fn [x] (* x 3)))
(my-special-multiplier 12)
; => 36
```

Clojure also offers another, more compact way to create anonymous functions. Here's what an anonymous function looks like:

```
#(* % 3)
```

Whoa, that looks weird. Go ahead and apply that weird-looking function:

```
(#(* % 3) 8)
; => 24
```

Here's an example of passing an anonymous function as an argument to map:

```
(map #(str "Hi, " %)
     ["Darth Vader" "Mr. Magoo"])
; => ("Hi, Darth Vader" "Hi, Mr. Magoo")
```

This strange-looking style of writing anonymous functions is made possible by a feature called *reader macros*. You'll learn all about those in Chapter 7. Right now, it's okay to learn how to use just these anonymous functions.

You can see that this syntax is definitely more compact, but it's also a little odd. Let's break it down. This kind of anonymous function looks a lot like a function call, except that it's preceded by a hash mark, #:

```
;; Function call
(* 8 3)

;; Anonymous function
#(* % 3)
```

This similarity allows you to more quickly see what will happen when this anonymous function is applied. "Oh," you can say to yourself, "this is going to multiply its argument by three."

As you may have guessed by now, the percent sign, %, indicates the argument passed to the function. If your anonymous function takes multiple arguments, you can distinguish them like this: %1, %2, %3, and so on. % is equivalent to %1:

```
(#(str %1 " and " %2) "cornbread" "butter beans")
; => "cornbread and butter beans"
```

You can also pass a rest parameter with %&:

```
(#(identity %&) 1 "blarg" :yip)
; => (1 "blarg" :yip)
```

In this case, you applied the identity function to the rest argument. Identity returns the argument it's given without altering it. Rest arguments are stored as lists, so the function application returns a list of all the arguments.

If you need to write a simple anonymous function, using this style is best because it's visually compact. On the other hand, it can easily become unreadable if you're writing a longer, more complex function. If that's the case, use fn.

Returning Functions

By now you've seen that functions can return other functions. The returned functions are *closures*, which means that they can access all the variables that were in scope when the function was created. Here's a standard example:

```
(defn inc-maker
  "Create a custom incrementor"
  [inc-by]
  #(+ % inc-by))

(def inc3 (inc-maker 3))

(inc3 7)
; => 10
```

; this isn't running for me, gives me this error:
IllegalArgumentException
Parameter declaration "inc-maker" should be a vector
clojure.core/assert-valid-fdecl (core.clj:7196)

Here, `inc-by` is in scope, so the returned function has access to it even when the returned function is used outside `inc-maker`.

Pulling It All Together

Okay! It's time to use your newfound knowledge for a noble purpose: smacking around hobbits! To hit a hobbit, you'll first model its body parts. Each body part will include its relative size to indicate how likely it is that that part will be hit. To avoid repetition, the hobbit model will include only entries for *left foot*, *left ear*, and so on. Therefore, you'll need a function to fully symmetrize the model, creating *right foot*, *right ear*, and so forth. Finally, you'll create a function that iterates over the body parts and randomly chooses the one hit. Along the way, you'll learn about a few new Clojure tools: `let` expressions, loops, and regular expressions. Fun!

The Shire's Next Top Model

For our hobbit model, we'll eschew such hobbit characteristics as joviality and mischievousness and focus only on the hobbit's tiny body. Here's the hobbit model:

```
(def asym-hobbit-body-parts [{:name "head" :size 3}
                             {:name "left-eye" :size 1}
                             {:name "left-ear" :size 1}
                             {:name "mouth" :size 1}
                             {:name "nose" :size 1}
                             {:name "neck" :size 2}
                             {:name "left-shoulder" :size 3}
                             {:name "left-upper-arm" :size 3}
                             {:name "chest" :size 10}
                             {:name "back" :size 10}
                             {:name "left-forearm" :size 3}
                             {:name "abdomen" :size 6}
                             {:name "left-kidney" :size 1}
                             {:name "left-hand" :size 2}
                             {:name "left-knee" :size 2}
                             {:name "left-thigh" :size 4}
                             {:name "left-lower-leg" :size 3}
                             {:name "left-achilles" :size 1}
                             {:name "left-foot" :size 2}])
```

This is a vector of maps. Each map has the name of the body part and relative size of the body part. (I know that only anime characters have eyes one-third the size of their head, but just go with it, okay?)

Conspicuously missing is the hobbit's right side. Let's fix that. Listing 3-1 is the most complex code you've seen so far, and it introduces some new ideas. But don't worry, because we'll examine it in great detail.

```
(defn matching-part
  [part]
  {:name (clojure.string/replace (:name part) #"^left-" "right-")
   :size (:size part)})

(defn symmetrize-body-parts
  "Expects a seq of maps that have a :name and :size"
  [asym-body-parts]
  (loop [remaining-asym-parts asym-body-parts
         final-body-parts []]
    (if (empty? remaining-asym-parts)
      final-body-parts
      (let [[part & remaining] remaining-asym-parts]
        (recur remaining
               (into final-body-parts
                     (set [part (matching-part part)])))))))
```

Listing 3-1: The `matching-part` and `symmetrize-body-parts` functions

When we call the function `symmetrize-body-parts` on `asym-hobbit-body-parts`, we get a fully symmetrical hobbit:

```
(symmetrize-body-parts asym-hobbit-body-parts)
; => [{:name "head", :size 3}
      {:name "left-eye", :size 1}
      {:name "right-eye", :size 1}
      {:name "left-ear", :size 1}
      {:name "right-ear", :size 1}
      {:name "mouth", :size 1}
      {:name "nose", :size 1}
      {:name "neck", :size 2}
      {:name "left-shoulder", :size 3}
      {:name "right-shoulder", :size 3}
      {:name "left-upper-arm", :size 3}
      {:name "right-upper-arm", :size 3}
      {:name "chest", :size 10}
      {:name "back", :size 10}
      {:name "left-forearm", :size 3}
      {:name "right-forearm", :size 3}
      {:name "abdomen", :size 6}
      {:name "left-kidney", :size 1}
      {:name "right-kidney", :size 1}
      {:name "left-hand", :size 2}
      {:name "right-hand", :size 2}
      {:name "left-knee", :size 2}
      {:name "right-knee", :size 2}
      {:name "left-thigh", :size 4}
      {:name "right-thigh", :size 4}
      {:name "left-lower-leg", :size 3}
      {:name "right-lower-leg", :size 3}
```

```
    {:name "left-achilles", :size 1}
    {:name "right-achilles", :size 1}
    {:name "left-foot", :size 2}
    {:name "right-foot", :size 2}]
```

Let's break down this code!

let

In the mass of craziness in Listing 3-1, you can see a form of the structure (`let ...`). Let's build up an understanding of `let` one example at a time, and then examine the full example from the program once we're familiar with all the pieces.

`let` binds names to values. You can think of `let` as short for *let it be*, which is also a beautiful Beatles song about programming. Here's an example:

```
(let [x 3]
  x)
; => 3

(def dalmatian-list
  ["Pongo" "Perdita" "Puppy 1" "Puppy 2"])
(let [dalmatians (take 2 dalmatian-list)]
  dalmatians)
; => ("Pongo" "Perdita")
```

In the first example, you bind the name `x` to the value `3`. In the second, you bind the name `dalmatians` to the result of the expression (`take 2 dalmatian-list`), which was the list (`"Pongo" "Perdita"`). `let` also introduces a new *scope*:

```
(def x 0)
(let [x 1] x)
; => 1
```

Here, you first bind the name `x` to the value `0` using `def`. Then, `let` creates a new scope in which the name `x` is bound to the value `1`. I think of scope as the context for what something refers to. For example, in the phrase "please clean up these butts," *butts* means something different depending on whether you're working in a maternity ward or on the custodial staff of a cigarette manufacturers convention. In this code snippet, you're saying, "I want `x` to be `0` in the global context, but within the context of this `let` expression, it should be `1`."

You can reference existing bindings in your `let` binding:

```
(def x 0)
(let [x (inc x)] x)
; => 1
```

In this example, the `x` in `(inc x)` refers to the binding created by `(def x 0)`. The resulting value is `1`, which is then bound to the name `x` within a new scope created by `let`. Within the confines of the `let` form, `x` refers to `1`, not `0`.

You can also use rest parameters in `let`, just like you can in functions:

```
(let [[pongo & dalmatians] dalmatian-list]
  [pongo dalmatians])
; => ["Pongo" ("Perdita" "Puppy 1" "Puppy 2")]
```

Notice that the value of a `let` form is the last form in its body that is evaluated. `let` forms follow all the destructuring rules introduced in "Calling Functions" on page 48. In this case, `[pongo & dalmatians]` destructured `dalmatian-list`, binding the string `"Pongo"` to the name `pongo` and the list of the rest of the dalmatians to `dalmatians`. The vector `[pongo dalmatians]` is the last expression in `let`, so it's the value of the `let` form.

`let` forms have two main uses. First, they provide clarity by allowing you to name things. Second, they allow you to evaluate an expression only once and reuse the result. This is especially important when you need to reuse the result of an expensive function call, like a network API call. It's also important when the expression has side effects.

Let's have another look at the `let` form in our symmetrizing function so we can understand exactly what's going on:

```
(let [[part & remaining] remaining-asym-parts]
  (recur remaining
         (into final-body-parts
               (set [part (matching-part part)]))))
```

; so this is destructuring part from remaining…
; using `recur` is how you ensure a process uses tail recursion (read on the Clojure spec)

This code tells Clojure, "Create a new scope. Within it, associate `part` with the first element of `remaining-asym-parts`. Associate `remaining` with the rest of the elements in `remaining-asym-parts`."

As for the body of the `let` expression, you'll learn about the meaning of `recur` in the next section. The function call

```
(into final-body-parts
  (set [part (matching-part part)]))
```

first tells Clojure, "Use the `set` function to create a set consisting of `part` and its matching part. Then use the function `into` to add the elements of that set to the vector `final-body-parts`." You create a set here to ensure you're adding unique elements to `final-body-parts` because `part` and `(matching-part part)` are sometimes the same thing, as you'll see in the upcoming section on regular expressions. Here's a simplified example:

```
(into [] (set [:a :a]))
; => [:a]
```

First, `(set [:a :a])` returns the set `#{:a}`, because sets don't contain duplicate elements. Then `(into [] #{:a})` returns the vector `[:a]`.

Back to `let`: notice that `part` is used multiple times in the body of the `let`. If we used the original expressions instead of using the names `part` and `remaining`, it would be a mess! Here's an example:

```
(recur (rest remaining-asym-parts)
       (into final-body-parts
             (set [(first remaining-asym-parts) (matching-part (first
remaining-asym-parts))])))
```

So, `let` is a handy way to introduce local names for values, which helps simplify the code.

loop

In our `symmetrize-body-parts` function we use `loop`, which provides another way to do recursion in Clojure. Let's look at a simple example:

```
(loop [iteration 0]
  (println (str "Iteration " iteration))
  (if (> iteration 3)
    (println "Goodbye!")
    (recur (inc iteration))))
; => Iteration 0
; => Iteration 1
; => Iteration 2
; => Iteration 3
; => Iteration 4
; => Goodbye!
```

The first line, `loop [iteration 0]`, begins the loop and introduces a binding with an initial value. On the first pass through the loop, `iteration` has a value of 0. Next, it prints a short message. Then, it checks the value of `iteration`. If the value is greater than 3, it's time to say Goodbye. Otherwise, we `recur`. It's as if `loop` creates an anonymous function with a parameter named `iteration`, and `recur` allows you to call the function from within itself, passing the argument `(inc iteration)`.

You could in fact accomplish the same thing by just using a normal function definition:

```
(defn recursive-printer
  ([]
     (recursive-printer 0))
  ([iteration]
     (println iteration)
     (if (> iteration 3)
       (println "Goodbye!")
       (recursive-printer (inc iteration)))))
(recursive-printer)
```

```
; => Iteration 0
; => Iteration 1
; => Iteration 2
; => Iteration 3
; => Iteration 4
; => Goodbye!
```

But as you can see, this is a bit more verbose. Also, `loop` has much better performance. In our symmetrizing function, we'll use `loop` to go through each element in the asymmetrical list of body parts.

Regular Expressions

Regular expressions are tools for performing pattern matching on text. The literal notation for a regular expression is to place the expression in quotes after a hash mark:

```
#"regular-expression"
```

In the function `matching-part` in Listing 3-1, `clojure.string/replace` uses the regular expression `#"^left-"` to match strings starting with `"left-"` in order to replace `"left-"` with `"right-"`. The carat, `^`, is how the regular expression signals that it will match the text `"left-"` only if it's at the beginning of the string, which ensures that something like `"cleft-chin"` won't match. You can test this with `re-find`, which checks whether a string matches the pattern described by a regular expression, returning the matched text or `nil` if there is no match: ;; regular-expression-find

```
(re-find #"^left-" "left-eye")
; => "left-"

(re-find #"^left-" "cleft-chin")
; => nil

(re-find #"^left-" "wongleblart")
; => nil
```

Here are a couple of examples of `matching-part` using a regex to replace `"left-"` with `"right-"`:

```
(defn matching-part
  [part]
  {:name (clojure.string/replace (:name part) #"^left-" "right-")
   :size (:size part)})
(matching-part {:name "left-eye" :size 1})
; => {:name "right-eye" :size 1}]

(matching-part {:name "head" :size 3})
; => {:name "head" :size 3}]
```

Notice that the name `"head"` is returned as is.

Symmetrizer

Now let's go back to the full symmetrizer and analyze it in more detail:

```
(def asym-hobbit-body-parts [{:name "head" :size 3}
                             {:name "left-eye" :size 1}
                             {:name "left-ear" :size 1}
                             {:name "mouth" :size 1}
                             {:name "nose" :size 1}
                             {:name "neck" :size 2}
                             {:name "left-shoulder" :size 3}
                             {:name "left-upper-arm" :size 3}
                             {:name "chest" :size 10}
                             {:name "back" :size 10}
                             {:name "left-forearm" :size 3}
                             {:name "abdomen" :size 6}
                             {:name "left-kidney" :size 1}
                             {:name "left-hand" :size 2}
                             {:name "left-knee" :size 2}
                             {:name "left-thigh" :size 4}
                             {:name "left-lower-leg" :size 3}
                             {:name "left-achilles" :size 1}
                             {:name "left-foot" :size 2}])


(defn matching-part
  [part]
  {:name (clojure.string/replace (:name part) #"^left-" "right-")
   :size (:size part)})

❶ (defn symmetrize-body-parts
    "Expects a seq of maps that have a :name and :size"
    [asym-body-parts]
❷   (loop [remaining-asym-parts asym-body-parts
           final-body-parts []]
❸     (if (empty? remaining-asym-parts)
        final-body-parts
❹       (let [[part & remaining] remaining-asym-parts]
❺         (recur remaining
                (into final-body-parts
                      (set [part (matching-part part)]))))))
```

The `symmetrize-body-parts` function (starting at ❶) employs a general strategy that is common in functional programming. Given a sequence (in this case, a vector of body parts and their sizes), the function continuously splits the sequence into a *head* and a *tail*. Then it processes the head, adds it to some result, and uses recursion to continue the process with the tail.

We begin looping over the body parts at ❷. The tail of the sequence will be bound to `remaining-asym-parts`. Initially, it's bound to the full sequence passed to the function: `asym-body-parts`. We also create a result sequence, `final-body-parts`; its initial value is an empty vector.

If `remaining-asym-parts` is empty at ❸, that means we've processed the entire sequence and can return the result, `final-body-parts`. Otherwise, at ❹ we split the list into a head, `part`, and tail, `remaining`.

At ❺, we recur with `remaining`, a list that gets shorter by one element on each iteration of the loop, and the `(into)` expression, which builds our vector of symmetrized body parts.

If you're new to this kind of programming, this code might take some time to puzzle out. Stick with it! Once you understand what's happening, you'll feel like a million bucks!

Better Symmetrizer with reduce

The pattern of *process each element in a sequence and build a result* is so common that there's a built-in function for it called `reduce`. Here's a simple example:

```
;; sum with reduce
(reduce + [1 2 3 4])
; => 10
```

This is like telling Clojure to do this:

```
(+ (+ (+ 1 2) 3) 4)
```

The `reduce` function works according to the following steps:

1. Apply the given function to the first two elements of a sequence. That's where `(+ 1 2)` comes from.
2. Apply the given function to the result and the next element of the sequence. In this case, the result of step 1 is `3`, and the next element of the sequence is `3` as well. So the final result is `(+ 3 3)`.
3. Repeat step 2 for every remaining element in the sequence.

`reduce` also takes an optional initial value. The initial value here is `15`:

```
(reduce + 15 [1 2 3 4])
```

If you provide an initial value, `reduce` starts by applying the given function to the initial value and the first element of the sequence rather than the first two elements of the sequence.

One detail to note is that, in these examples, `reduce` takes a collection of elements, `[1 2 3 4]`, and returns a single number. Although programmers often use `reduce` this way, you can also use `reduce` to return an even larger collection than the one you started with, as we're trying to do with `symmetrize-body-parts`. `reduce` abstracts the task "process a collection

and build a result," which is agnostic about the type of result returned. To further understand how reduce works, here's one way that you could implement it:

```
(defn my-reduce
  ([f initial coll]
   (loop [result initial
          remaining coll]
     (if (empty? remaining)
       result
       (recur (f result (first remaining)) (rest remaining)))))
  ([f [head & tail]]
   (my-reduce f head tail)))
```

We could reimplement our symmetrizer as follows:

```
(defn better-symmetrize-body-parts
  "Expects a seq of maps that have a :name and :size"
  [asym-body-parts]
  (reduce (fn [final-body-parts part]
            (into final-body-parts (set [part (matching-part part)])))
          []
          asym-body-parts))
```

Groovy! One immediately obvious advantage of using reduce is that you write less code overall. The anonymous function you pass to reduce focuses only on processing an element and building a result. The reason is that reduce handles the lower-level machinery of keeping track of which elements have been processed and deciding whether to return a final result or to recur.

Using reduce is also more expressive. If readers of your code encounter loop, they won't be sure exactly what the loop is doing without reading all of the code. But if they see reduce, they'll immediately know that the purpose of the code is to process the elements of a collection to build a result.

Finally, by abstracting the reduce process into a function that takes another function as an argument, your program becomes more composable. You can pass the reduce function as an argument to other functions, for example. You could also create a more generic version of symmetrize-body-parts, say, expand-body-parts. This could take an *expander* function in addition to a list of body parts and would let you model more than just hobbits. For example, you could have a spider expander that could multiply the numbers of eyes and legs. I'll leave it up to you to write that because I am evil.

Hobbit Violence

My word, this is truly Clojure for the Brave and True! To put the capstone on your work, here's a function that determines which part of a hobbit is hit:

```
(defn hit
  [asym-body-parts]
```

```
(let [sym-parts (❶better-symmetrize-body-parts asym-body-parts)
      ❷body-part-size-sum (reduce + (map :size sym-parts))
      target (rand body-part-size-sum)]
  ❸(loop [[part & remaining] sym-parts
          accumulated-size (:size part)]
    (if (> accumulated-size target)
      part
      (recur remaining (+ accumulated-size (:size (first remaining))))))))
```

`hit` works by taking a vector of asymmetrical body parts, symmetrizing it at ❶, and then summing the sizes of the parts at ❷. Once we sum the sizes, it's like each number from 1 through `body-part-size-sum` corresponds to a body part; 1 might correspond to the left eye, and 2, 3, 4 might correspond to the head. This makes it so when you hit a body part (by choosing a random number in this range), the likelihood that a particular body part is hit will depend on the size of the body part.

Finally, one of these numbers is randomly chosen, and then we use `loop` at ❸ to find and return the body part that corresponds to the number. The loop does this by keeping track of the accumulated sizes of parts that we've checked and checking whether the accumulated size is greater than the target. I visualize this process as lining up the body parts with a row of numbered slots. After I line up a body part, I ask myself, "Have I reached the target yet?" If I have, that means the body part I just lined up was the one hit. Otherwise, I just keep lining up those parts.

For example, say that your list of parts is *head*, *left eye*, and *left hand*, like in Figure 3-1. After taking the first part, the head, the accumulated size is 3. The body part is hit if the accumulated size is greater than the target, so if the target is 0, 1, or 2, then the head was hit. Otherwise, you take the next part, the left eye, and increase the accumulated size to 4, yielding a hit if the target is 3. Similarly, the left hand gets hit if the target is 4 or 5.

Figure 3-1: Body parts correspond to ranges of numbers and get hit if the target falls within that range.

Here are some sample runs of the `hit` function:

```
(hit asym-hobbit-body-parts)
; => {:name "right-upper-arm", :size 3}

(hit asym-hobbit-body-parts)
; => {:name "chest", :size 10}

(hit asym-hobbit-body-parts)
; => {:name "left-eye", :size 1}
```

Oh my god, that poor hobbit! You monster!

Summary

This chapter gave you a whirlwind tour of how to *do stuff* in Clojure. You now know how to represent information using strings, numbers, maps, keywords, vectors, lists, and sets, and how to name these representations with `def` and `let`. You've learned about how flexible functions are and how to create your own functions. Also, you've been introduced to Clojure's philosophy of simplicity, including its uniform syntax and its emphasis on using large libraries of functions on primitive data types.

Chapter 4 will take you through a detailed examination of Clojure's core functions, and Chapter 5 explains the functional programming mindset. This chapter has shown you how to write Clojure code—the next two will show you how to write Clojure *well*.

At this point I recommend, with every fiber of my being, that you start writing code. There is no better way to solidify your Clojure knowledge. The Clojure Cheat Sheet (*http://clojure.org/cheatsheet/*) is a great reference that lists all the built-in functions that operate on the data structures covered in this chapter.

The following exercises will really tickle your brain. If you'd like to test your new skills even more, try some Project Euler challenges at *http://www.projecteuler.net/*. You could also check out 4Clojure (*http://www.4clojure.com/problems/*), an online set of Clojure problems designed to test your knowledge. Just write something!

Exercises

These exercises are meant to be a fun way to test your Clojure knowledge and to learn more Clojure functions. The first three can be completed using only the information presented in this chapter, but the last three will require you to use functions that haven't been covered so far. Tackle the last three if you're really itching to write more code and explore Clojure's standard library. If you find the exercises too difficult, revisit them after reading Chapters 4 and 5—you'll find them much easier.

1. Use the `str`, `vector`, `list`, `hash-map`, and `hash-set` functions.
2. Write a function that takes a number and adds 100 to it.
3. Write a function, `dec-maker`, that works exactly like the function `inc-maker` except with subtraction:

```
(def dec9 (dec-maker 9))
(dec9 10)
; => 1
```

4. Write a function, `mapset`, that works like `map` except the return value is a set:

```
(mapset inc [1 1 2 2])
; => #{2 3}
```

5. Create a function that's similar to `symmetrize-body-parts` except that it has to work with weird space aliens with radial symmetry. Instead of two eyes, arms, legs, and so on, they have five.
6. Create a function that generalizes `symmetrize-body-parts` and the function you created in Exercise 5. The new function should take a collection of body parts and the number of matching body parts to add. If you're completely new to Lisp languages and functional programming, it probably won't be obvious how to do this. If you get stuck, just move on to the next chapter and revisit the problem later.

4

CORE FUNCTIONS IN DEPTH

If you're a huge fan of the angsty, teenager-centric, quasi–soap opera *The Vampire Diaries* like I am, you'll remember the episode where the lead protagonist, Elena, starts to question her pale, mysterious crush's behavior: "Why did he instantly vanish without a trace when I scraped my knee?" and "How come his face turned into a grotesque mask of death when I nicked my finger?" and so on.

You might be asking yourself similar questions if you've started playing with Clojure's core functions. "Why did `map` return a list when I gave it a vector?" and "How come `reduce` treats my map like a list of vectors?" and so on. (With Clojure, though, you're at least spared from contemplating the profound existential horror of being a 17-year-old for eternity.)

In this chapter, you'll learn about Clojure's deep, dark, bloodthirsty, supernatur—**cough** I mean, in this chapter, you'll learn about Clojure's

underlying concept of *programming to abstractions* and about the sequence and collection abstractions. You'll also learn about *lazy sequences*. This will give you the grounding you need to read the documentation for functions you haven't used before and to understand what's happening when you give them a try.

Next, you'll get more experience with the functions you'll be reaching for the most. You'll learn how to work with lists, vectors, maps, and sets with the functions `map`, `reduce`, `into`, `conj`, `concat`, `some`, `filter`, `take`, `drop`, `sort`, `sort-by`, and `identity`. You'll also learn how to create new functions with `apply`, `partial`, and `complement`. All this information will help you understand how to do things the Clojure way, and it will give you a solid foundation for writing your own code as well as for reading and learning from others' projects.

Finally, you'll learn how to parse and query a CSV of vampire data to determine what nosferatu lurk in your hometown.

Programming to Abstractions

To understand programming to abstractions, let's compare Clojure to a language that wasn't built with that principle in mind: Emacs Lisp (elisp). In elisp, you can use the `mapcar` function to derive a new list, which is similar to how you use `map` in Clojure. However, if you want to map over a hash map (similar to Clojure's map data structure) in elisp, you'll need to use the `maphash` function, whereas in Clojure you can still just use `map`. In other words, elisp uses two different, data structure–specific functions to implement the *map* operation, but Clojure uses only one. You can also call `reduce` on a map in Clojure, whereas elisp doesn't provide a function for reducing a hash map.

The reason is that Clojure defines `map` and `reduce` functions in terms of the *sequence abstraction*, not in terms of specific data structures. As long as a data structure responds to the core sequence operations (the functions `first`, `rest`, and `cons`, which we'll look at more closely in a moment), it will work with `map`, `reduce`, and oodles of other sequence functions for free. This is what Clojurists mean by programming to abstractions, and it's a central tenet of Clojure philosophy.

I think of abstractions as named collections of operations. If you can perform all of an abstraction's operations on an object, then that object is an instance of the abstraction. I think this way even outside of programming. For example, the *battery* abstraction includes the operation "connect a conducting medium to its anode and cathode," and the operation's output is *electrical current*. It doesn't matter if the battery is made out of lithium or out of potatoes. It's a battery as long as it responds to the set of operations that define *battery*.

Similarly, `map` doesn't care about how lists, vectors, sets, and maps are implemented. It only cares about whether it can perform sequence operations on them. Let's look at how `map` is defined in terms of the sequence abstraction so you can understand programming to abstractions in general.

Treating Lists, Vectors, Sets, and Maps as Sequences

If you think about the `map` operation independently of any programming language, or even of programming altogether, its essential behavior is to derive a new sequence y from an existing sequence x using a function f such that $y_1 = f(x_1)$, $y_2 = f(x_2)$, . . . $y_n = f(x_n)$. Figure 4-1 illustrates how you might visualize a mapping applied to a sequence.

Figure 4-1: Visualizing a mapping

The term *sequence* here refers to a collection of elements organized in linear order, as opposed to, say, an unordered collection or a graph without a *before-and-after* relationship between its nodes. Figure 4-2 shows how you might visualize a sequence, in contrast to the other two collections mentioned.

Figure 4-2: Sequential and nonsequential collections

Absent from this description of mapping and sequences is any mention of lists, vectors, or other concrete data structures. Clojure is designed to allow us to think and program in such abstract terms as much as possible, and it does this by implementing functions in terms of data structure abstractions. In this case, `map` is defined in terms of the sequence abstraction. In conversation, you would say `map`, `reduce`, and other sequence functions *take a sequence*

or even *take a seq*. In fact, Clojurists usually use *seq* instead of *sequence*, using terms like *seq functions* and *the seq library* to refer to functions that perform sequential operations. Whether you use *sequence* or *seq*, you're indicating that the data structure in question will be treated as a sequence and that what it actually is in its truest heart of hearts doesn't matter in this context.

If the core sequence functions `first`, `rest`, and `cons` work on a data structure, you can say the data structure *implements* the sequence abstraction. Lists, vectors, sets, and maps all implement the sequence abstraction, so they all work with `map`, as shown here:

```
(defn titleize
  [topic]
  (str topic " for the Brave and True"))

(map titleize ["Hamsters" "Ragnarok"])
; => ("Hamsters for the Brave and True" "Ragnarok for the Brave and True")

(map titleize '("Empathy" "Decorating"))
; => ("Empathy for the Brave and True" "Decorating for the Brave and True")

(map titleize #{"Elbows" "Soap Carving"})
; => ("Elbows for the Brave and True" "Soap Carving for the Brave and True")

(map #(titleize (second %)) {:uncomfortable-thing "Winking"})
; => ("Winking for the Brave and True")
```

The first two examples show that `map` works identically with vectors and lists. The third example shows that `map` can work with unsorted sets. In the fourth example, you must call `second` on the anonymous function's argument before title-izing it because the argument is a map. I'll explain why soon, but first let's look at the three functions that define the sequence abstraction.

first, rest, and cons

In this section, we'll take a quick detour into JavaScript to implement a linked list and three core functions: `first`, `rest`, and `cons`. After those three core functions are implemented, I'll show how you to build `map` with them.

The point is to appreciate the distinction between the seq abstraction in Clojure and the concrete implementation of a linked list. It doesn't matter how a particular data structure is implemented: when it comes to using seq functions on a data structure, all Clojure asks is "can I `first`, `rest`, and `cons` it?" If the answer is yes, you can use the seq library with that data structure.

In a linked list, nodes are linked in a linear sequence. Here's how you might create one in JavaScript. In this snippet, `next` is null because this is the last node in the list:

```
var node3 = {
  value: "last",
  next: null
};
```

In this snippet, `node2`'s `next` points to `node3`, and `node1`'s `next` points to `node2`; that's the "link" in "linked list":

```
var node2 = {
  value: "middle",
  next: node3
};

var node1 = {
  value: "first",
  next: node2
};
```

Graphically, you could represent this list as shown in Figure 4-3.

Figure 4-3: A linked list

You can perform three core functions on a linked list: `first`, `rest`, and `cons`. `first` returns the value for the requested node, `rest` returns the remaining values after the requested node, and `cons` adds a new node with the given value to the beginning of the list. After those are implemented, you can implement `map`, `reduce`, `filter`, and other seq functions on top of them.

The following code shows how we would implement and use `first`, `rest`, and `cons` with our JavaScript node example, as well as how to use them to return specific nodes and derive a new list. Note that the parameter of `first` and `rest` is named *node*. This might be confusing because you might say, "Ain't I getting the first element of a *list?*" Well, you operate on the elements of a list one node at a time!

```
var first = function(node) {
  return node.value;
};

var rest = function(node) {
  return node.next;
};
```

```
var cons = function(newValue, node) {
  return {
    value: newValue,
    next: node
  };
};

first(node1);
// => "first"

first(rest(node1));
// => "middle"

first(rest(rest(node1)));
// => "last"

var node0 = cons("new first", node1);
first(node0);
// => "new first"

first(rest(node0));
// => "first"
```

As noted previously, you can implement `map` in terms of `first`, `rest`, and `cons`:

```
var map = function (list, transform) {
  if (list === null) {
    return null;
  } else {
    return cons(transform(first(list)), map(rest(list), transform));
  }
}
```

This function transforms the first element of the list and then calls itself again on the rest of the list until it reaches the end (a null). Let's see it in action! In this example, you're mapping the list that begins with `node1`, returning a new list where the string `" mapped!"` is appended to each node's value. Then you're using `first` to return the first node's value:

```
first(
  map(node1, function (val) { return val + " mapped!"})
);

// => "first mapped!"
```

So here's the cool thing: because `map` is implemented completely in terms of `cons`, `first`, and `rest`, you could actually pass it any data structure and it would work as long as `cons`, `first`, and `rest` work on that data structure.

Here's how they might work for an array:

```
var first = function (array) {
  return array[0];
}

var rest = function (array) {
  var sliced = array.slice(1, array.length);
  if (sliced.length == 0) {
    return null;
  } else {
    return sliced;
  }
}

var cons = function (newValue, array) {
  return [newValue].concat(array);
}


var list = ["Transylvania", "Forks, WA"];
map(list, function (val) { return val + " mapped!"})
// => ["Transylvania mapped!", "Forks, WA mapped!"]
```

This code snippet defines `first`, `rest`, and `cons` in terms of JavaScript's array functions. Meanwhile, `map` continues referencing functions named `first`, `rest`, and `cons`, so now it works on `array`. So, if you can just implement `first`, `rest`, and `cons`, you get `map` for free along with the aforementioned oodles of other functions.

Abstraction Through Indirection

At this point, you might object that I'm just kicking the can down the road because we're still left with the problem of how a function like `first` is able to work with different data structures. Clojure does this using two forms of indirection. In programming, *indirection* is a generic term for the mechanisms a language employs so that one name can have multiple, related meanings. In this case, the name `first` has multiple, data structure–specific meanings. Indirection is what makes abstraction possible.

Polymorphism is one way that Clojure provides indirection. I don't want to get lost in the details, but basically, polymorphic functions dispatch to different function bodies based on the type of the argument supplied. (It's not so different from how multiple-arity functions dispatch to different function bodies based on the number of arguments you provide.)

NOTE *Clojure has two constructs for defining polymorphic dispatch: the host platform's* interface *construct and platform-independent* protocols. *But it's not necessary to understand how these work when you're just getting started. I'll cover protocols in Chapter 13.*

When it comes to sequences, Clojure also creates indirection by doing a kind of lightweight type conversion, producing a data structure that works with an abstraction's functions. Whenever Clojure expects a sequence—for example, when you call `map`, `first`, `rest`, or `cons`—it calls the `seq` function on the data structure in question to obtain a data structure that allows for `first`, `rest`, and `cons`:

```
(seq '(1 2 3))
; => (1 2 3)

(seq [1 2 3])
; => (1 2 3)

(seq #{1 2 3})
; => (1 2 3)

(seq {:name "Bill Compton" :occupation "Dead mopey guy"})
; => ([:name "Bill Compton"] [:occupation "Dead mopey guy"])
```

There are two notable details here. First, `seq` always returns a value that looks and behaves like a list; you'd call this value a *sequence* or *seq*. Second, the seq of a map consists of two-element key-value vectors. That's why `map` treats your maps like lists of vectors! You can see this in the `"Bill Compton"` example. I wanted to point out this example in particular because it might be surprising and confusing. It was for me when I first started using Clojure. Knowing these underlying mechanisms will spare you from the kind of frustration and general mopiness often exhibited by male vampires trying to retain their humanity.

You can convert the seq back into a map by using `into` to stick the result into an empty map (you'll look at `into` closely later):

```
(into {} (seq {:a 1 :b 2 :c 3}))
; => {:a 1, :c 3, :b 2}     ; is there a reason the :b and :c keywords switched their order?
```

So, Clojure's sequence functions use `seq` on their arguments. The sequence functions are defined in terms of the sequence abstraction, using `first`, `rest`, and `cons`. As long as a data structure implements the sequence abstraction, it can use the extensive seq library, which includes such superstar functions as `reduce`, `filter`, `distinct`, `group-by`, and dozens more.

The takeaway here is that it's powerful to focus on what we can *do* with a data structure and to ignore, as much as possible, its implementation. Implementations rarely matter in and of themselves. They're just a means to an end. In general, programming to abstractions gives you power by letting you use libraries of functions on different data structure regardless of how those data structures are implemented.

Seq Function Examples

Clojure's seq library is full of useful functions that you'll use all the time. Now that you have a deeper understanding of Clojure's sequence abstraction, let's look at these functions in detail. If you're new to Lisp and functional programming, these examples will be surprising and delightful.

map

You've seen many examples of `map` by now, but this section shows `map` doing two new tasks: taking multiple collections as arguments and taking a collection of functions as an argument. It also highlights a common `map` pattern: using keywords as the mapping function.

So far, you've only seen examples of `map` operating on one collection. In the following code, the collection is the vector `[1 2 3]`:

```
(map inc [1 2 3])
; => (2 3 4)
```

However, you can also give `map` multiple collections. Here's a simple example to show how this works:

```
(map str ["a" "b" "c"] ["A" "B" "C"])
; => ("aA" "bB" "cC")
```

It's as if `map` does the following:

```
(list (str "a" "A") (str "b" "B") (str "c" "C"))
```

When you pass `map` multiple collections, the elements of the first collection (`["a" "b" "c"]`) will be passed as the first argument of the mapping function (`str`), the elements of the second collection (`["A" "B" "C"]`) will be passed as the second argument, and so on. Just be sure that your mapping function can take a number of arguments equal to the number of collections you're passing to `map`.

The following example shows how you could use this capability if you were a vampire trying to curb your human consumption. You have two vectors, one representing human intake in liters and another representing critter intake for the past four days. The `unify-diet-data` function takes a single day's data for both human and critter feeding and unifies the two into a single map:

```
(def human-consumption   [8.1 7.3 6.6 5.0])
(def critter-consumption [0.0 0.2 0.3 1.1])
(defn unify-diet-data
  [human critter]
  {:human human
   :critter critter})
```

```
(map unify-diet-data human-consumption critter-consumption)
; => ({:human 8.1, :critter 0.0}
      {:human 7.3, :critter 0.2}
      {:human 6.6, :critter 0.3}
      {:human 5.0, :critter 1.8})
```

Good job laying off the human!

Another fun thing you can do with `map` is pass it a collection of functions. You could use this if you wanted to perform a set of calculations on different collections of numbers, like so:

```
(def sum #(reduce + %))
(def avg #(/ (sum %) (count %)))
(defn stats
  [numbers]
  (map #(% numbers) [sum count avg]))

(stats [3 4 10])
; => (17 3 17/3)

(stats [80 1 44 13 6])
; => (144 5 144/5)
```

In this example, the `stats` function iterates over a vector of functions, applying each function to `numbers`.

Additionally, Clojurists often use `map` to retrieve the value associated with a keyword from a collection of map data structures. Because keywords can be used as functions, you can do this succinctly. Here's an example:

```
(def identities
  [{:alias "Batman" :real "Bruce Wayne"}
   {:alias "Spider-Man" :real "Peter Parker"}
   {:alias "Santa" :real "Your mom"}
   {:alias "Easter Bunny" :real "Your dad"}])

(map :real identities)
; => ("Bruce Wayne" "Peter Parker" "Your mom" "Your dad")
```

(If you are five, then I apologize profusely.)

reduce

Chapter 3 showed how `reduce` processes each element in a sequence to build a result. This section shows a couple of other ways to use it that might not be obvious.

The first use is to transform a map's values, producing a new map with the same keys but with updated values:

```
(reduce (fn [new-map [key val]]
          (assoc new-map key (inc val)))
```

```
        {}
        {:max 30 :min 10})
; => {:max 31, :min 11}
```

In this example, `reduce` treats the argument `{:max 30 :min 10}` as a sequence of vectors, like `([:max 30] [:min 10])`. Then, it starts with an empty map (the second argument) and builds it up using the first argument, an anonymous function. It's as if `reduce` does this:

```
(assoc (assoc {} :max (inc 30))
       :min (inc 10))
```

Another use for `reduce` is to filter out keys from a map based on their value. In the following example, the anonymous function checks whether the value of a key-value pair is greather than 4. If it isn't, then the key-value pair is filtered out. In the map `{:human 4.1 :critter 3.9}`, 3.9 is less than 4, so the `:critter` key and its 3.9 value are filtered out.

```
(reduce (fn [new-map [key val]]
          (if (> val 4)
            (assoc new-map key val)
            new-map))
        {}
        {:human 4.1
         :critter 3.9})
; => {:human 4.1}
```

The takeaway here is that `reduce` is a more flexible function than it first appears. Whenever you want to derive a new value from a seqable data structure, `reduce` will usually be able to do what you need. If you want an exercise that will really blow your hair back, try implementing `map` using `reduce`, and then do the same for `filter` and `some` after you read about them later in this chapter.

take, drop, take-while, and drop-while

`take` and `drop` both take two arguments: a number and a sequence. `take` returns the first *n* elements of the sequence, whereas `drop` returns the sequence with the first *n* elements removed:

```
(take 3 [1 2 3 4 5 6 7 8 9 10])
; => (1 2 3)

(drop 3 [1 2 3 4 5 6 7 8 9 10])
; => (4 5 6 7 8 9 10)
```

Their cousins `take-while` and `drop-while` are a bit more interesting. Each takes a *predicate function* (a function whose return value is evaluated for truth or falsity) to determine when it should stop taking or dropping.

Suppose, for example, that you had a vector representing entries in your "food" journal. Each entry has the year, month, day, and what you ate. To preserve space, we'll only include a few entries:

```
(def food-journal
  [{:month 1 :day 1 :human 5.3 :critter 2.3}
   {:month 1 :day 2 :human 5.1 :critter 2.0}
   {:month 2 :day 1 :human 4.9 :critter 2.1}
   {:month 2 :day 2 :human 5.0 :critter 2.5}
   {:month 3 :day 1 :human 4.2 :critter 3.3}
   {:month 3 :day 2 :human 4.0 :critter 3.8}
   {:month 4 :day 1 :human 3.7 :critter 3.9}
   {:month 4 :day 2 :human 3.7 :critter 3.6}])
```

With `take-while`, you can retrieve just January's and February's data. `take-while` traverses the given sequence (in this case, `food-journal`), applying the predicate function to each element.

This example uses the anonymous function `#(< (:month %) 3)` to test whether the journal entry's month is out of range:

```
(take-while #(< (:month %) 3) food-journal)
; => ({:month 1 :day 1 :human 5.3 :critter 2.3}
      {:month 1 :day 2 :human 5.1 :critter 2.0}
      {:month 2 :day 1 :human 4.9 :critter 2.1}
      {:month 2 :day 2 :human 5.0 :critter 2.5})
```

When `take-while` reaches the first March entry, the anonymous function returns `false`, and `take-while` returns a sequence of every element it tested until that point.

The same idea applies with `drop-while` except that it keeps dropping elements until one tests true:

```
(drop-while #(< (:month %) 3) food-journal)
; => ({:month 3 :day 1 :human 4.2 :critter 3.3}
      {:month 3 :day 2 :human 4.0 :critter 3.8}
      {:month 4 :day 1 :human 3.7 :critter 3.9}
      {:month 4 :day 2 :human 3.7 :critter 3.6})
```

By using `take-while` and `drop-while` together, you can get data for just February and March:

```
(take-while #(< (:month %) 4)
            (drop-while #(< (:month %) 2) food-journal))
; => ({:month 2 :day 1 :human 4.9 :critter 2.1}
      {:month 2 :day 2 :human 5.0 :critter 2.5}
      {:month 3 :day 1 :human 4.2 :critter 3.3}
      {:month 3 :day 2 :human 4.0 :critter 3.8})
```

This example uses `drop-while` to get rid of the January entries, and then it uses `take-while` on the result to keep taking entries until it reaches the first April entry.

filter and some

Use `filter` to return all elements of a sequence that test true for a predicate function. Here are the journal entries where human consumption is less than five liters:

```
(filter #(< (:human %) 5) food-journal)
; => ({:month 2 :day 1 :human 4.9 :critter 2.1}
      {:month 3 :day 1 :human 4.2 :critter 3.3}
      {:month 3 :day 2 :human 4.0 :critter 3.8}
      {:month 4 :day 1 :human 3.7 :critter 3.9}
      {:month 4 :day 2 :human 3.7 :critter 3.6})
```

You might be wondering why we didn't just use `filter` in the `take-while` and `drop-while` examples earlier. Indeed, `filter` would work for that too. Here we're grabbing the January and February data, just like in the `take-while` example:

```
(filter #(< (:month %) 3) food-journal)
; => ({:month 1 :day 1 :human 5.3 :critter 2.3}
      {:month 1 :day 2 :human 5.1 :critter 2.0}
      {:month 2 :day 1 :human 4.9 :critter 2.1}
      {:month 2 :day 2 :human 5.0 :critter 2.5})
```

This use is perfectly fine, but `filter` can end up processing all of your data, which isn't always necessary. Because the food journal is already sorted by date, we know that `take-while` will return the data we want without having to examine any of the data we won't need. Therefore, `take-while` can be more efficient.

Often, you want to know whether a collection contains any values that test true for a predicate function. The `some` function does that, returning the first truthy value (any value that's not `false` or `nil`) returned by a predicate function:

```
(some #(> (:critter %) 5) food-journal)
; => nil

(some #(> (:critter %) 3) food-journal)
; => true
```

You don't have any food journal entries where you consumed more than five liters from critter sources, but you do have at least one where you consumed more than three liters. Notice that the return value in the second example is `true` and not the actual entry that produced the true value. The reason is that the anonymous function `#(> (:critter %) 3)` returns `true` or `false`. Here's how you could return the entry:

```
(some #(and (> (:critter %) 3) %) food-journal)
; => {:month 3 :day 1 :human 4.2 :critter 3.3}
```

;; yeah I was wondering about that.
;; throw in AND, which actually returns the value,
;; if you care what it is.

Here, a slightly different anonymous function uses `and` to first check whether the condition `(> (:critter %) 3)` is true, and then returns the entry when the condition is indeed true.

sort and sort-by

You can sort elements in ascending order with `sort`:

```
(sort [3 1 2])
; => (1 2 3)
```

If your sorting needs are more complicated, you can use `sort-by`, which allows you to apply a function (sometimes called a *key function*) to the elements of a sequence and use the values it returns to determine the sort order. In the following example, which is taken from *http://clojuredocs.org/*, `count` is the key function:

```
(sort-by count ["aaa" "c" "bb"])   ; wow, awesome example
; => ("c" "bb" "aaa")
```

If you were sorting using `sort`, the elements would be sorted in alphabetical order, returning `("aaa" "bb" "c")`. Instead, the result is `("c" "bb" "aaa")` because you're sorting by `count` and the count of `"c"` is 1, `"bb"` is 2, and `"aaa"` is 3.

concat

Finally, `concat` simply appends the members of one sequence to the end of another:

```
(concat [1 2] [3 4])
; => (1 2 3 4)
```

Lazy Seqs

As you saw earlier, `map` first calls `seq` on the collection you pass to it. But that's not the whole story. Many functions, including `map` and `filter`, return a *lazy seq*. A lazy seq is a seq whose members aren't computed until you try to access them. Computing a seq's members is called *realizing* the seq. Deferring the computation until the moment it's needed makes your programs more efficient, and it has the surprising benefit of allowing you to construct infinite sequences.

Demonstrating Lazy Seq Efficiency

To see lazy seqs in action, pretend that you're part of a modern-day task force whose purpose is to identify vampires. Your intelligence agents tell you that there is only one active vampire in your city, and they've helpfully

narrowed down the list of suspects to a million people. Your boss gives you a list of one million Social Security numbers and shouts, "Get it done, McFishwich!" ; hah

Thankfully, you are in possession of a Vampmatic 3000 computifier, the state-of-the-art device for vampire identification. Because the source code for this vampire-hunting technology is proprietary, I've stubbed it out to simulate the time it would take to perform this task. Here is a subset of a vampire database:

```
(def vampire-database
  {0 {:makes-blood-puns? false, :has-pulse? true  :name "McFishwich"}
   1 {:makes-blood-puns? false, :has-pulse? true  :name "McMackson"}
   2 {:makes-blood-puns? true,  :has-pulse? false :name "Damon Salvatore"}
   3 {:makes-blood-puns? true,  :has-pulse? true  :name "Mickey Mouse"}})

(defn vampire-related-details
  [social-security-number]
  (Thread/sleep 1000)
  (get vampire-database social-security-number))

(defn vampire?
  [record]
  (and (:makes-blood-puns? record)
       (not (:has-pulse? record))
       record))

(defn identify-vampire
  [social-security-numbers]
  (first (filter vampire?
                 (map vampire-related-details social-security-numbers))))
```

;; I thought you needed a DO block to perform
;; multiple actions?

You have a function, `vampire-related-details`, which takes one second to look up an entry from the database. Next, you have a function, `vampire?`, which returns a record if it passes the vampire test; otherwise, it returns `false`. Finally, `identify-vampire` maps Social Security numbers to database records and then returns the first record that indicates vampirism.

To show how much time it takes to run these functions, you can use the `time` operation. When you use `time`, your code behaves exactly as it would if you didn't use `time`, but with one exception: a report of the elapsed time is printed. Here's an example:

```
(time (vampire-related-details 0))
; => "Elapsed time: 1001.042 msecs"
; => {:name "McFishwich", :makes-blood-puns? false, :has-pulse? true}
```

The first printed line reports the time taken by the given operation—in this case, 1,001.042 milliseconds. The second is the return value, which is your database record in this case. The return value is exactly the same as it would have been if you hadn't used `time`.

A nonlazy implementation of `map` would first have to apply `vampire-related-details` to every member of `social-security-numbers` before passing

the result to `filter`. Because you have one million suspects, this would take one million seconds, or 12 days, and half your city would be dead by then! Of course, if it turns out that the only vampire is the last suspect in the record, it will still take that much time with the lazy version, but at least there's a good chance that it won't.

Because `map` is lazy, it doesn't actually apply `vampire-related-details` to Social Security numbers until you try to access the mapped element. In fact, `map` returns a value almost instantly:

```
(time (def mapped-details (map vampire-related-details (range 0 1000000))))
; => "Elapsed time: 0.049 msecs"
; => #'user/mapped-details
```

In this example, `range` returns a lazy sequence consisting of the integers from 0 to 999,999. Then, `map` returns a lazy sequence that is associated with the name `mapped-details`. Because `map` didn't actually apply `vampire-related-details` to any of the elements returned by `range`, the entire operation took barely any time—certainly less than 12 days.

You can think of a lazy seq as consisting of two parts: a recipe for how to realize the elements of a sequence and the elements that have been realized so far. When you use `map`, the lazy seq it returns doesn't include any realized elements yet, but it does have the recipe for generating its elements. Every time you try to access an unrealized element, the lazy seq will use its recipe to generate the requested element.

In the previous example, `mapped-details` is unrealized. Once you try to access a member of `mapped-details`, it will use its recipe to generate the element you've requested, and you'll incur the one-second-per-database-lookup cost:

```
(time (first mapped-details))
; => "Elapsed time: 32030.767 msecs"
; => {:name "McFishwich", :makes-blood-puns? false, :has-pulse? true}
```

This operation took about 32 seconds. That's much better than one million seconds, but it's still 31 seconds more than we would have expected. After all, you're only trying to access the very first element, so it should have taken only one second.

The reason it took 32 seconds is that Clojure *chunks* its lazy sequences, which just means that whenever Clojure has to realize an element, it preemptively realizes some of the next elements as well. In this example, you wanted only the very first element of `mapped-details`, but Clojure went ahead and prepared the next 31 as well. Clojure does this because it almost always results in better performance.

Thankfully, lazy seq elements need to be realized only once. Accessing the first element of `mapped-details` again takes almost no time:

```
(time (first mapped-details))
; => "Elapsed time: 0.022 msecs"
; => {:name "McFishwich", :makes-blood-puns? false, :has-pulse? true}
```

With all this newfound knowledge, you can efficiently mine the vampire database to find the fanged culprit:

```
(time (identify-vampire (range 0 1000000)))
"Elapsed time: 32019.912 msecs"
; => {:name "Damon Salvatore", :makes-blood-puns? true, :has-pulse? false}
```

Ooh! That's why Damon makes those creepy puns!

Infinite Sequences

One cool, useful capability that lazy seqs give you is the ability to construct infinite sequences. So far, you've only worked with lazy sequences generated from vectors or lists that terminated. However, Clojure comes with a few functions to create infinite sequences. One easy way to create an infinite sequence is with `repeat`, which creates a sequence whose every member is the argument you pass:

```
(concat (take 8 (repeat "na")) ["Batman!"])
; => ("na" "na" "na" "na" "na" "na" "na" "na" "Batman!")
```

In this case, you create an infinite sequence whose every element is the string `"na"`, then use that to construct a sequence that may or not provoke nostalgia.

You can also use `repeatedly`, which will call the provided function to generate each element in the sequence:

```
(take 3 (repeatedly (fn [] (rand-int 10))))
; => (1 4 0)
```

Here, the lazy sequence returned by `repeatedly` generates every new element by calling the anonymous function `(fn [] (rand-int 10))`, which returns a random integer between 0 and 9. If you run this in your REPL, your result will most likely be different from this one.

A lazy seq's recipe doesn't have to specify an endpoint. Functions like `first` and `take`, which realize the lazy seq, have no way of knowing what will come next in a seq, and if the seq keeps providing elements, well, they'll just keep taking them. You can see this if you construct your own infinite sequence:

```
(defn even-numbers
  ([] (even-numbers 0))
  ([n] (cons n (lazy-seq (even-numbers (+ n 2))))))

(take 10 (even-numbers))
; => (0 2 4 6 8 10 12 14 16 18)
```

; i get it. if passed no args, first branch calls the function initializing it with 0

This example is a bit mind-bending because of its use of recursion. It helps to remember that `cons` returns a new list with an element appended to the given list:

```
(cons 0 '(2 4 6))
; => (0 2 4 6)
```

(Incidentally, Lisp programmers call it *consing* when they use the `cons` function.)

In `even-numbers`, you're consing to a lazy list, which includes a recipe (a function) for the next element (as opposed to consing to a fully realized list).

And that covers lazy seqs! Now you know everything there is to know about the sequence abstraction, and we can turn to the collection abstraction!

The Collection Abstraction

The collection abstraction is closely related to the sequence abstraction. All of Clojure's core data structures—vectors, maps, lists, and sets—take part in both abstractions.

The sequence abstraction is about operating on members individually, whereas the collection abstraction is about the data structure as a whole. For example, the collection functions `count`, `empty?`, and `every?` aren't about any individual element; they're about the whole:

```
(empty? [])
; => true

(empty? ["no!"])
; => false
```

Practically speaking, you'll rarely consciously say, "Okay, self! You're working with the collection as a whole now. Think in terms of the collection abstraction!" Nevertheless, it's useful to know these concepts that underlie the functions and data structures you're using.

Now we'll examine two common collection functions—`into` and `conj`—whose similarities can be a bit confusing.

into

One of the most important collection functions is `into`. As you now know, many seq functions return a seq rather than the original data structure. You'll probably want to convert the return value back into the original value, and `into` lets you do that:

```
(map identity {:sunlight-reaction "Glitter!"})
; => ([:sunlight-reaction "Glitter!"])

(into {} (map identity {:sunlight-reaction "Glitter!"}))
; => {:sunlight-reaction "Glitter!"}
```

Here, the `map` function returns a sequential data structure after being given a map data structure, and into converts the seq back into a map.

This will work with other data structures as well:

```
(map identity [:garlic :sesame-oil :fried-eggs])
; => (:garlic :sesame-oil :fried-eggs)

(into [] (map identity [:garlic :sesame-oil :fried-eggs]))
; => [:garlic :sesame-oil :fried-eggs]
```

Here, in the first line, `map` returns a seq, and we use `into` in the second line to convert the result back to a vector.

In the following example, we start with a vector with two identical entries, `map` converts it to a list, and then we use `into` to stick the values into a set.

```
(map identity [:garlic-clove :garlic-clove])
; => (:garlic-clove :garlic-clove)

(into #{} (map identity [:garlic-clove :garlic-clove]))
; => #{:garlic-clove}
```

Because sets only contain unique values, the set ends up with just one value in it.

The first argument of `into` doesn't have to be empty. Here, the first example shows how you can use `into` to add elements to a map, and the second shows how you can add elements to a vector.

```
(into {:favorite-emotion "gloomy"} [[:sunlight-reaction "Glitter!"]])
; => {:favorite-emotion "gloomy" :sunlight-reaction "Glitter!"}

(into ["cherry"] '("pine" "spruce"))
; => ["cherry" "pine" "spruce"]
```

And, of course, both arguments can be the same type. In this next example, both arguments are maps, whereas all the previous examples had arguments of different types. It works as you'd expect, returning a new map with the elements of the second map added to the first:

```
(into {:favorite-animal "kitty"} {:least-favorite-smell "dog"
                                  :relationship-with-teenager "creepy"})
; => {:favorite-animal "kitty"
      :relationship-with-teenager "creepy"  ; this seems like a very common use case
      :least-favorite-smell "dog"}
```

If `into` were asked to describe its strengths at a job interview, it would say, "I'm great at taking two collections and adding all the elements from the second to the first."

conj

`conj` also adds elements to a collection, but it does it in a slightly different way:

```
(conj [0] [1])
; => [0 [1]]
```

Whoopsie! Looks like it added the entire vector `[1]` to `[0]`. Compare this with `into`:

```
(into [0] [1])
; => [0 1]
```

Here's how we'd do the same with `conj`:

```
(conj [0] 1)
; => [0 1]
```

Notice that the number 1 is passed as a scalar (singular, non-collection) value, whereas `into`'s second argument must be a collection.

You can supply as many elements to add with `conj` as you want, and you can also add to other collections like maps:

```
(conj [0] 1 2 3 4)
; => [0 1 2 3 4]

(conj {:time "midnight"} [:place "ye olde cemetarium"])
; => {:place "ye olde cemetarium" :time "midnight"}
```

`conj` and `into` are so similar that you could even define `conj` in terms of `into`:

```
(defn my-conj
  [target & additions]
  (into target additions))

(my-conj [0] 1 2 3)
; => [0 1 2 3]
```

This kind of pattern isn't that uncommon. You'll often see two functions that do the same thing, except one takes a rest parameter (`conj`) and one takes a seqable data structure (`into`).

Function Functions

Learning to take advantage of Clojure's ability to accept functions as arguments and return functions as values is really fun, even if it takes some getting used to.

Two of Clojure's functions, `apply` and `partial`, might seem especially weird because they both accept *and* return functions. Let's unweird them.

apply

`apply` *explodes* a seqable data structure so it can be passed to a function that expects a rest parameter. For example, `max` takes any number of arguments and returns the greatest of all the arguments. Here's how you'd find the greatest number:

```
(max 0 1 2)
; => 2
```

But what if you want to find the greatest element of a vector? You can't just pass the vector to `max`:

```
(max [0 1 2])
; => [0 1 2]
```

This doesn't return the greatest element in the vector because `max` returns the greatest of all the arguments passed to it, and in this case you're only passing it a vector containing all the numbers you want to compare, rather than passing in the numbers as separate arguments. `apply` is perfect for this situation:

```
(apply max [0 1 2])
; => 2
```

By using `apply`, it's as if you called `(max 0 1 2)`. You'll often use `apply` like this, exploding the elements of a collection so that they get passed to a function as separate arguments.

Remember how we defined `conj` in terms of `into` earlier? Well, we can also define `into` in terms of `conj` by using `apply`:

```
(defn my-into
  [target additions]
  (apply conj target additions))

(my-into [0] [1 2 3])
; => [0 1 2 3]
```

This call to `my-into` is equivalent to calling `(conj [0] 1 2 3)`.

partial

`partial` takes a function and any number of arguments. It then returns a new function. When you call the returned function, it calls the original function with the original arguments you supplied it along with the new arguments.

Here's an example:

```
(def add10 (partial + 10))
(add10 3)
; => 13
```

```
(add10 5)
; => 15

(def add-missing-elements
  (partial conj ["water" "earth" "air"]))

(add-missing-elements "unobtainium" "adamantium")
; => ["water" "earth" "air" "unobtainium" "adamantium"]
```

So when you call `add10`, it calls the original function and arguments `(+ 10)` and tacks on whichever arguments you call `add10` with. To help clarify how `partial` works, here's how you might define it:

```
(defn my-partial
  [partialized-fn & args]
  (fn [& more-args]
    (apply partialized-fn (into args more-args))))

(def add20 (my-partial + 20))
(add20 3)
; => 23
```

In this example, the value of `add20` is the anonymous function returned by `my-partial`. The anonymous function is defined like this:

```
(fn [& more-args]
  (apply + (into [20] more-args)))
```

In general, you want to use partials when you find you're repeating the same combination of function and arguments in many different contexts. This toy example shows how you could use `partial` to specialize a logger, creating a `warn` function:

```
(defn lousy-logger
  [log-level message]
  (condp = log-level
    :warn (clojure.string/lower-case message)
    :emergency (clojure.string/upper-case message)))

(def warn (partial lousy-logger :warn))

(warn "Red light ahead")
; => "red light ahead"
```

Calling `(warn "Red light ahead")` here is identical to calling `(lousy-logger :warn "Red light ahead")`.

complement

Earlier you created the `identify-vampire` function to find one vampire amid a million people. What if you wanted to create a function to find all humans?

Perhaps you want to send them thank-you cards for not being an undead predator. Here's how you could do it:

```
(defn identify-humans
  [social-security-numbers]
  (filter #(not (vampire? %))
          (map vampire-related-details social-security-numbers)))
```

Look at the first argument to `filter`, `#(not (vampire? %))`. It's so common to want the *complement* (the negation) of a Boolean function that there's a function, `complement`, for that:

```
(def not-vampire? (complement vampire?))
(defn identify-humans
  [social-security-numbers]
  (filter not-vampire?
          (map vampire-related-details social-security-numbers)))
```

Here's how you might implement `complement`:

```
(defn my-complement
  [fun]
  (fn [& args]
    (not (apply fun args))))

(def my-pos? (complement neg?))
(my-pos? 1)
; => true

(my-pos? -1)
; => false
```

As you can see, `complement` is a humble function. It does one little thing and does it well. `complement` made it trivial to create a `not-vampire?` function, and anyone reading the code could understand the code's intention.

This won't MapReduce terabytes of data for you or anything like that, but it does demonstrate the power of higher-order functions. They allow you to build up libraries of utility functions in a way that is impossible in some languages. In aggregate, these utility functions make your life a lot easier.

A Vampire Data Analysis Program for the FWPD

To pull everything together, let's write the beginnings of a sophisticated vampire data analysis program for the Forks, Washington Police Department (FWPD).

The FWPD has a fancy new database technology called *CSV (comma-separated values)*. Your job is to parse this state-of-the-art CSV and analyze

it for potential vampires. We'll do that by filtering on each suspect's *glitter index*, a 0–10 prediction of the suspect's vampireness developed by some teenage girl. Go ahead and create a new Leiningen project for your tool:

```
lein new app fwpd
```

Under the new *fwpd* directory, create a file named *suspects.csv* and enter contents like the following:

```
Edward Cullen,10
Bella Swan,0
Charlie Swan,0
Jacob Black,3
Carlisle Cullen,6
```

Now it's time to get your hands dirty by building up the *fwpd/src/fwpd/core.clj* file. I recommend that you start a new REPL session so you can try things out as you go along. In Emacs you can do this by opening *fwpd/src/fwpd/core.clj* and running **M-x `cider-restart`**. Once the REPL is started, delete the contents of *core.clj*, and then add this:

```
(ns fwpd.core)
(def filename "suspects.csv")
```

The first line establishes the namespace, and the second just makes it a tiny bit easier to refer to the CSV you created. You can do a quick sanity check in your REPL by compiling your file (**C-c C-k** in Emacs) and running this:

```
(slurp filename)
; => "Edward Cullen,10\nBella Swan,0\nCharlie Swan,0\nJacob Black,3\nCarlisle Cullen,6"
```

If the `slurp` function doesn't return the preceding string, try restarting your REPL session with *core.clj* open.

Next, add this to *core.clj*:

```
❶ (def vamp-keys [:name :glitter-index])

❷ (defn str->int
    [str]
    (Integer. str))

❸ (def conversions {:name identity
                    :glitter-index str->int})

❹ (defn convert
    [vamp-key value]
    ((get conversions vamp-key) value))
```

Ultimately, you'll end up with a sequence of maps that look like `{:name "Edward Cullen" :glitter-index 10}`, and the preceding definitions help

you get there. First, vamp-keys ❶ is a vector of the keys that you'll soon use to create vampire maps. Next, the function str->int ❷ converts a string to an integer. The map conversions ❸ associates a conversion function with each of the vamp keys. You don't need to transform the name at all, so its conversion function is identity, which just returns the argument passed to it. The glitter index is converted to an integer, so its conversion function is str->int. Finally, the convert function ❹ takes a vamp key and a value, and returns the converted value. Here's an example:

```
(convert :glitter-index "3")
; => 3
```

Now add this to your file:

```
(defn parse
  "Convert a CSV into rows of columns"
  [string]
  (map #(clojure.string/split % #",")
       (clojure.string/split string #"\n")))
```

The parse function takes a string and first splits it on the newline character to create a seq of strings. Next, it maps over the seq of strings, splitting each one on the comma character. Try running parse on your CSV:

```
(parse (slurp filename))
; => (["Edward Cullen" "10"] ["Bella Swan" "0"] ["Charlie Swan" "0"]
      ["Jacob Black" "3"] ["Carlisle Cullen" "6"])
```

The next bit of code takes the seq of vectors and combines it with your vamp keys to create maps:

```
(defn mapify
  "Return a seq of maps like {:name \"Edward Cullen\" :glitter-index 10}"
  [rows]
  (map (fn [unmapped-row]
         (reduce (fn [row-map [vamp-key value]]
                   (assoc row-map vamp-key (convert vamp-key value)))
                 {}
                 (map vector vamp-keys unmapped-row)))
       rows))
```

In this function, map transforms each row—vectors like ["Bella Swan" 0]—into a map by using reduce in a manner similar to the first example in "reduce" on page 80. First, map creates a seq of key-value pairs like ([:name "Bella Swan"] [:glitter-index] 0). Then, reduce builds up a map by associating a vamp key with a converted vamp value into row-map. Here's the first row mapified:

```
(first (mapify (parse (slurp filename))))
; => {:glitter-index 10, :name "Edward Cullen"}
```

Finally, add this `glitter-filter` function:

```
(defn glitter-filter
  [minimum-glitter records]
  (filter #(>= (:glitter-index %) minimum-glitter) records))
```

This takes fully mapified vampire records and filters out those with a `:glitter-index` less than the provided `minimum-glitter`:

```
(glitter-filter 3 (mapify (parse (slurp filename))))
({:name "Edward Cullen", :glitter-index 10}
 {:name "Jacob Black", :glitter-index 3}
 {:name "Carlisle Cullen", :glitter-index 6})
```

Et voilà! You are now one step closer to fulfilling your dream of being a supernatural-creature-hunting vigilante. You better go round up those sketchy characters!

Summary

In this chapter, you learned that Clojure emphasizes programming to abstractions. The sequence abstraction deals with operating on the individual elements of a sequence, and seq functions often convert their arguments to a seq and return a lazy seq. Lazy evaluation improves performance by delaying computations until they're needed. The other abstraction you learned about, the collection abstraction, deals with data structures as a whole. Finally, the most important thing you learned is that you should never trust someone who sparkles in sunlight.

Exercises

The vampire analysis program you now have is already decades ahead of anything else on the market. But how could you make it better? I suggest trying the following:

1. Turn the result of your glitter filter into a list of names.
2. Write a function, `append`, which will append a new suspect to your list of suspects.
3. Write a function, `validate`, which will check that `:name` and `:glitter-index` are present when you `append`. The `validate` function should accept two arguments: a map of keywords to validating functions, similar to `conversions`, and the record to be validated.
4. Write a function that will take your list of maps and convert it back to a CSV string. You'll need to use the `clojure.string/join` function.

Good luck, McFishwich!

5

FUNCTIONAL PROGRAMMING

So far, you've focused on becoming familiar with the tools that Clojure provides: immutable data structures, functions, abstractions, and so on. In this chapter, you'll learn how to think about your programming tasks in a way that makes the best use of those tools. You'll begin integrating your experience into a new functional programming mindset.

The core concepts you'll learn include: what pure functions are and why they're useful; how to work with immutable data structures and why they're superior to their mutable cousins; how disentangling data and functions gives you more power and flexibility; and why it's powerful to program to a small set of data abstractions. Once you shove all this knowledge into your brain matter, you'll have an entirely new approach to problem solving!

After going over these topics, you'll put everything you've learned to use by writing a terminal-based game inspired by an ancient, mystic mind-training device found in Cracker Barrel restaurants across America: Peg Thing!

Pure Functions: What and Why

Except for `println` and `rand`, all the functions you've used up till now have been pure functions. What makes them pure functions, and why does it matter? A function is pure if it meets two qualifications:

- It always returns the same result if given the same arguments. This is called *referential transparency*, and you can add it to your list of $5 programming terms.
- It can't cause any side effects. That is, the function can't make any changes that are observable outside the function itself—for example, by changing an externally accessible mutable object or writing to a file.

These qualities make it easier for you to reason about your programs because the functions are completely isolated, unable to impact other parts of your system. When you use them, you don't have to ask yourself, "What could I break by calling this function?" They're also consistent: you'll never need to figure out why passing a function the same arguments results in different return values, because that will never happen.

Pure functions are as stable and problem free as arithmetic (when was the last time you fretted over adding two numbers?). They're stupendous little bricks of functionality that you can confidently use as the foundation of your program. Let's look at referential transparency and lack of side effects in more detail to see exactly what they are and how they're helpful.

Pure Functions Are Referentially Transparent

To return the same result when called with the same argument, pure functions rely only on 1) their own arguments and 2) immutable values to determine their return value. Mathematical functions, for example, are referentially transparent:

```
(+ 1 2)
; => 3
```

If a function relies on an immutable value, it's referentially transparent. The string `", Daniel-san"` is immutable, so the following function is also referentially transparent:

```
(defn wisdom
  [words]
  (str words ", Daniel-san"))

(wisdom "Always bathe on Fridays")
; => "Always bathe on Fridays, Daniel-san"
```

By contrast, the following functions do not yield the same result with the same arguments; therefore, they are not referentially transparent. Any function that relies on a random number generator cannot be referentially transparent:

```
(defn year-end-evaluation
  []
  (if (> (rand) 0.5)
    "You get a raise!"
    "Better luck next year!"))
```

If your function reads from a file, it's not referentially transparent because the file's contents can change. The following function, `analyze-file`, is not referentially transparent, but the function `analysis` is:

```
(defn analyze-file
  [filename]
  (analysis (slurp filename)))

(defn analysis
  [text]
  (str "Character count: " (count text)))
```

When using a referentially transparent function, you never have to consider what possible external conditions could affect the return value of the function. This is especially important if your function is used multiple places or if it's nested deeply in a chain of function calls. In both cases, you can rest easy knowing that changes to external conditions won't cause your code to break.

Another way to think about this is that reality is largely referentially transparent. If you think of gravity as a function, then gravitational force is the return value of calling that function on two objects. Therefore, the next time you're in a programming interview, you can demonstrate your functional programming knowledge by knocking everything off your interviewer's desk. (This also demonstrates that you know how to apply a function over a collection.)

Pure Functions Have No Side Effects

To perform a side effect is to change the association between a name and its value within a given scope. Here is an example in JavaScript:

```
var haplessObject = {
  emotion: "Carefree!"
};

var evilMutator = function(object){
  object.emotion = "So emo :'(";
}
```

```
evilMutator(haplessObject);
haplessObject.emotion;
// => "So emo :'("
```

Of course, your program has to have some side effects. It writes to a disk, which changes the association between a filename and a collection of disk sectors; it changes the RGB values of your monitor's pixels; and so on. Otherwise, there'd be no point in running it.

Side effects are potentially harmful, however, because they introduce uncertainty about what the names in your code are referring to. This leads to situations where it's very difficult to trace why and how a name came to be associated with a value, which makes it hard to debug the program. When you call a function that doesn't have side effects, you only have to consider the relationship between the input and the output. You don't have to worry about other changes that could be rippling through your system.

Functions with side effects, on the other hand, place more of a burden on your mind grapes: now you have to worry about how the world is affected when you call the function. Not only that, but every function that depends on a side-effecting function gets infected by this worry; it, too, becomes another component that requires extra care and thought as you build your program.

If you have any significant experience with a language like Ruby or JavaScript, you've probably run into this problem. As an object gets passed around, its attributes somehow change, and you can't figure out why. Then you have to buy a new computer because you've chucked yours out the window. If you've read anything about object-oriented design, you know that a lot of writing has been devoted to strategies for managing state and reducing side effects for just this reason.

For all these reasons, it's a good idea to look for ways to limit the use of side effects in your code. Lucky for you, Clojure makes your job easier by going to great lengths to limit side effects—all of its core data structures are immutable. You cannot change them in place, no matter how hard you try! However, if you're unfamiliar with immutable data structures, you might feel like your favorite tool has been taken from you. How can you *do* anything without side effects? Well, that's what the next section is all about! How about that segue, eh? Eh?

Living with Immutable Data Structures

Immutable data structures ensure that your code won't have side effects. As you now know with all your heart, this is a good thing. But how do you get anything done without side effects?

Recursion Instead of for/while

Raise your hand if you've ever written something like this in JavaScript:

```
var wrestlers = getAlligatorWrestlers();
var totalBites = 0;
```

```
var l = wrestlers.length;

for(var i=0; i < l; i++){
  totalBites += wrestlers[i].timesBitten;
}
```

Or this:

```
var allPatients = getArkhamPatients();
var analyzedPatients = [];
var l = allPatients.length;

for(var i=0; i < l; i++){
  if(allPatients[i].analyzed){
    analyzedPatients.push(allPatients[i]);
  }
}
```

Notice that both examples induce side effects on the looping variable `i`, as well as a variable outside the loop (`totalBites` in the first example and `analyzedPatients` in the second). Using side effects this way—mutating *internal* variables—is pretty much harmless. You're creating new values, as opposed to changing an object you've received from elsewhere in your program.

But Clojure's core data structures don't even allow these harmless mutations. So what can you do instead? First, ignore the fact that you could easily use `map` and `reduce` to accomplish the preceding work. In these situations—iterating over some collection to build a result—the functional alternative to mutation is recursion.

Let's look at the first example, building a sum. Clojure has no assignment operator. You can't associate a new value with a name without creating a new scope:

```
(def great-baby-name "Rosanthony")
great-baby-name
; => "Rosanthony"

(let [great-baby-name "Bloodthunder"]
  great-baby-name)
; => "Bloodthunder"

great-baby-name
; => "Rosanthony"
```

In this example, you first bind the name `great-baby-name` to `"Rosanthony"` within the global scope. Next, you introduce a new scope with `let`. Within

that scope, you bind `great-baby-name` to `"Bloodthunder"`. Once Clojure finishes evaluating the `let` expression, you're back in the global scope, and `great-baby-name` evaluates to `"Rosanthony"` once again.

Clojure lets you work around this apparent limitation with recursion. The following example shows the general approach to recursive problem solving:

```
(defn sum
❶  ([vals] (sum vals 0))
   ([vals accumulating-total]
❷     (if (empty? vals)
        accumulating-total
        (sum (rest vals) (+ (first vals) accumulating-total)))))
```

This function takes two arguments, a collection to process (`vals`) and an accumulator (`accumulating-total`), and it uses arity overloading (covered in Chapter 3) to provide a default value of `0` for `accumulating-total` at ❶.

Like all recursive solutions, this function checks the argument it's processing against a base condition. In this case, we check whether `vals` is empty at ❷. If it is, we know that we've processed all the elements in the collection, so we return `accumulating-total`.

If `vals` isn't empty, it means we're still working our way through the sequence, so we recursively call `sum` passing it two arguments: the *tail* of vals with `(rest vals)` and the sum of the first element of `vals` plus the accumulating total with `(+ (first vals) accumulating-total)`. In this way, we build up `accumulating-total` and at the same time reduce `vals` until it reaches the base case of an empty collection.

Here's what the recursive function call might look like if we separate out each time it recurs:

```
(sum [39 5 1]) ; single-arity body calls two-arity body
(sum [39 5 1] 0)
(sum [5 1] 39)
(sum [1] 44)
(sum [] 45) ; base case is reached, so return accumulating-total
; => 45
```

Each recursive call to `sum` creates a new scope where `vals` and `accumulating-total` are bound to different values, all without needing to alter the values originally passed to the function or perform any internal mutation. As you can see, you can get along fine without mutation.

Note that you should generally use `recur` when doing recursion for performance reasons. The reason is that Clojure doesn't provide tail call optimization, a topic I will never bring up again! (Check out this URL for more information: *http://en.wikipedia.org/wiki/Tail_call*.) So here's how you'd do this with `recur`:

; unlike Scheme, which does provide support for tail call recursion

```
(defn sum
  ([vals]
     (sum vals 0))
  ([vals accumulating-total]
```

```
    (if (empty? vals)
      accumulating-total
      (recur (rest vals) (+ (first vals) accumulating-total)))))
```

;; errata: thousands or millions OF values

Using `recur` isn't that important if you're recursively operating on a small collection, but if your collection contains thousands or millions values, you will definitely need to whip out `recur` so you don't blow up your program with a stack overflow.

One last thing! You might be saying, "Wait a minute—what if I end up creating thousands of intermediate values? Doesn't this cause the program to thrash because of garbage collection or whatever?"

Very good question, eagle-eyed reader! The answer is no! The reason is that, behind the scenes, Clojure's immutable data structures are implemented using *structural sharing*, which is totally beyond the scope of this book. It's kind of like Git! Read this great article if you want to know more: *http://hypirion.com/musings/understanding-persistent-vector-pt-1*.

Function Composition Instead of Attribute Mutation

Another way you might be used to using mutation is to build up the final state of an object. In the following Ruby example, the `GlamourShotCaption` object uses mutation to clean input by removing trailing spaces and capitalizing `"lol"`:

```
class GlamourShotCaption
  attr_reader :text
  def initialize(text)
    @text = text
    clean!
  end

  private
  def clean!
    text.trim!
    text.gsub!(/lol/, "LOL")
  end
end

best = GlamourShotCaption.new("My boa constrictor is so sassy lol!  ")
best.text
; => "My boa constrictor is so sassy LOL!"
```

In this code, the class `GlamourShotCaption` encapsulates the knowledge of how to clean a glamour shot caption. On creating a `GlamourShotCaption` object, you assign text to an instance variable and progressively mutate it.

Listing 5-1 shows how you might do this in Clojure:

```
(require '[clojure.string :as s])
(defn clean
  [text]
  (s/replace (s/trim text) #"lol" "LOL"))
```

```
(clean "My boa constrictor is so sassy lol!  ")
; => "My boa constrictor is so sassy LOL!"
```

Listing 5-1: Using function composition to modify a glamour shot caption

In the first line, we use `require` to access the string function library (I'll discuss this function and related concepts in Chapter 6). Otherwise, the code is easy peasy. No mutation required. Instead of progressively mutating an object, the `clean` function works by passing an immutable value, `text`, to a pure function, `s/trim`, which returns an immutable value (`"My boa constrictor is so sassy lol!"`; the spaces at the end of the string have been trimmed). That value is then passed to the pure function `s/replace`, which returns another immutable value (`"My boa constrictor is so sassy LOL!"`).

Combining functions like this—so that the return value of one function is passed as an argument to another—is called *function composition*. In fact, this isn't so different from the previous example, which used recursion, because recursion continually passes the result of a function to another function; it just happens to be the same function. In general, functional programming encourages you to build more complex functions by combining simpler functions.

This comparison also starts to reveal some limitations of object-oriented programming (OOP). In OOP, one of the main purposes of classes is to protect against unwanted modification of private data—something that isn't necessary with immutable data structures. You also have to tightly couple methods with classes, thus limiting the reusability of the methods. In the Ruby example, you have to do extra work to reuse the `clean!` method. In Clojure, `clean` will work on any string at all. By both a) decoupling functions and data, and b) programming to a small set of abstractions, you end up with more reusable, composable code. You gain power and lose nothing.

Going beyond immediately practical concerns, the differences between the way you write object-oriented and functional code point to a deeper difference between the two mindsets. Programming is about manipulating data for your own nefarious purposes (as much as you can say it's *about* anything). In OOP, you think about data as something you can embody in an object, and you poke and prod it until it looks right. During this process, your original data is lost forever unless you're very careful about preserving it. By contrast, in functional programming you think of data as unchanging, and you derive new data from existing data. During this process, the original data remains safe and sound. In the preceding Clojure example,

the original caption doesn't get modified. It's safe in the same way that numbers are safe when you add them together; you don't somehow transform 4 into 7 when you add 3 to it.

Once you are confident using immutable data structures to get stuff done, you'll feel even more confident because you won't have to worry about what dirty code might be getting its greasy paws on your precious, mutable variables. It'll be great!

Cool Things to Do with Pure Functions

You can derive new functions from existing functions in the same way that you derive new data from existing data. You've already seen one function, `partial`, that creates new functions. This section introduces you to two more functions, `comp` and `memoize`, which rely on referential transparency, immutability, or both.

comp

It's always safe to compose pure functions like we just did in the previous section, because you only need to worry about their input/output relationship. Composing functions is so common that Clojure provides a function, `comp`, for creating a new function from the composition of any number of functions. Here's a simple example:

```
((comp inc *) 2 3)
; => 7
```

Here, you create an anonymous function by composing the `inc` and `*` functions. Then, you immediately apply this function to the arguments `2` and `3`. The function multiplies the numbers 2 and 3 and then increments the result. Using math notation, you'd say that, in general, using `comp` on the functions $f_1, f_2, \ldots f_n$, creates a new function g such that $g(x_1, x_2, \ldots x_n)$ equals $f_1(f_2(f_n(x_1, x_2, \ldots x_n)))$. One detail to note here is that the first function applied—`*` in the code shown here—can take any number of arguments, whereas the remaining functions must be able to take only one argument.

Here's an example that shows how you could use `comp` to retrieve character attributes in role-playing games:

```
(def character
  {:name "Smooches McCutes"
   :attributes {:intelligence 10
                :strength 4
                :dexterity 5}})
(def c-int (comp :intelligence :attributes))
(def c-str (comp :strength :attributes))
(def c-dex (comp :dexterity :attributes))
                ; very cool and simple
(c-int character)
; => 10
```

```
(c-str character)
; => 4

(c-dex character)
; => 5
```

In this example, you created three functions that help you look up a character's attributes. Instead of using `comp`, you could just have written something like this for each attribute:

```
(fn [c] (:strength (:attributes c)))
```

But `comp` is more elegant because it uses less code to convey more meaning. When you see `comp`, you immediately know that the resulting function's purpose is to combine existing functions in a well-known way.

What do you do if one of the functions you want to compose needs to take more than one argument? You wrap it in an anonymous function. Have a look at this next snippet, which calculates the number of spell slots your character has based on her intelligence attribute:

```
(defn spell-slots
  [char]
  (int (inc (/ (c-int char) 2))))

(spell-slots character)
; => 6
```

First, you divide intelligence by two, then you add one, and then you use the `int` function to round down. Here's how you could do the same thing with `comp`:

```
(def spell-slots-comp (comp int inc #(/ % 2) c-int))
```

To divide by two, all you needed to do was wrap the division in an anonymous function.

Clojure's `comp` function can compose any number of functions. To get a hint of how it does this, here's an implementation that composes just two functions:

```
(defn two-comp
  [f g]
  (fn [& args]
    (f (apply g args))))
```

I encourage you to evaluate this code and use `two-comp` to compose two functions! Also, try reimplementing Clojure's `comp` function so you can compose any number of functions.

memoize

Another cool thing you can do with pure functions is memoize them so that Clojure remembers the result of a particular function call. You can do this because, as you learned earlier, pure functions are referentially transparent. For example, `+` is referentially transparent. You can replace

```
(+ 3 (+ 5 8))
```

with

```
(+ 3 13)
```

or

```
16
```

and the program will have the same behavior.

Memoization lets you take advantage of referential transparency by storing the arguments passed to a function and the return value of the function. That way, subsequent calls to the function with the same arguments can return the result immediately. This is especially useful for functions that take a lot of time to run. For example, in this unmemoized function, the result is returned after one second:

```
(defn sleepy-identity
  "Returns the given value after 1 second"
  [x]
  (Thread/sleep 1000)
  x)
(sleepy-identity "Mr. Fantastico")
; => "Mr. Fantastico" after 1 second

(sleepy-identity "Mr. Fantastico")
; => "Mr. Fantastico" after 1 second
```

However, if you create a new, memoized version of `sleepy-identity` with `memoize`, only the first call waits one second; every subsequent function call returns immediately:

```
(def memo-sleepy-identity (memoize sleepy-identity))
(memo-sleepy-identity "Mr. Fantastico")
; => "Mr. Fantastico" after 1 second

(memo-sleepy-identity "Mr. Fantastico")
; => "Mr. Fantastico" immediately
```

Pretty cool! From here on out, `memo-sleepy-identity` will not incur the initial one-second cost when called with `"Mr. Fantastico"`. This implementation could be useful for functions that are computationally intensive or that make network requests.

Peg Thing

It's that time! Time to build a terminal implementation of Peg Thing using everything you've learned so far: immutable data structures, lazy sequences, pure functions, recursion—everything! Doing this will help you understand how to combine these concepts and techniques to solve larger problems. Most important, you'll learn how to model the changes that result from each move a player makes without having to mutate objects like you would in OOP.

To build the game, you'll first learn the game's mechanics and how to start and play the program. Then, I'll explain the code's organization. Finally, I'll walk through each function.

You can find the complete code repository for Peg Thing at *https://www.nostarch.com/clojure/*.

Playing

As mentioned earlier, Peg Thing is based on the secret mind-sharpening tool passed down from ye olden days and is now distributed by Cracker Barrel. ; wait but what happened to Bill's wife?

If you're not familiar with the game, here are the mechanics. You start out with a triangular board consisting of holes filled with pegs, and one hole has a missing a peg, as shown in Figure 5-1.

Figure 5-1: The initial setup for Peg Thing

The object of the game is to remove as many pegs as possible. You do this by *jumping* over pegs. In Figure 5-2, peg A jumps over peg B into the empty hole, and you remove peg B from the board.

Figure 5-2: Jump a peg to remove it from the board.

To start Peg Thing, download the code, and then run `lein run` in your terminal while in the *pegthing* directory. A prompt appears that looks like this:

```
Get ready to play Peg Thing!
How many rows? [5]
```

Now you can enter the number of rows the board will have, using `5` as the default. If you want five rows, just press ENTER (otherwise, type a number and press ENTER). You'll then see this:

```
Here's your board:
       a0
      b0 c0
    d0 e0 f0
   g0 h0 i0 j0
 k0 l0 m0 n0 o0
Remove which peg? [e]
```

Each letter identifies a position on the board. The number `0` (which should be blue, but if it's not, it's no big deal) indicates that a position is filled. Before gameplay begins, one peg must be empty, so the prompt asks you to enter the position of the first to peg to remove. The default is the center peg, `e`, but you can choose a different one. After you remove the peg, you'll see this:

```
Here's your board:
       a0
      b0 c0
    d0 e- f0
   g0 h0 i0 j0
 k0 l0 m0 n0 o0
Move from where to where? Enter two letters:
```

Notice that the `e` position now has a dash, `-` (which should be red, but if it's not, it's no big deal). The dash indicates that the position is empty. To

move, you enter the position of the peg you want to *pick up* followed by the position of the empty position that you want to place it in. If you enter `le`, for example, you'll get this:

```
Here's your board:
       a0
      b0 c0
    d0 e0 f0
   g0 h- i0 j0
 k0 l- m0 n0 o0
Move from where to where? Enter two letters:
```

You've moved the peg from `l` to `e`, jumping over `h` and removing its peg according to the rule shown in Figure 5-2. The game continues to prompt you for moves until no moves are available, whereupon it prompts you to play again.

Code Organization

The program has to handle four major tasks, and the source code is organized accordingly, with the functions for each of these tasks grouped together:

1. Creating a new board
2. Returning a board with the result of the player's move
3. Representing a board textually
4. Handling user interaction

Two more points about the organization: First, the code has a basic *architecture*, or conceptual organization, of two layers. The top layer consists of the functions for handling user interaction. These functions produce all of the program's side effects, printing out the board and presenting prompts for player interaction. The functions in this layer use the functions in the bottom layer to create a new board, make moves, and create a textual representation, but the functions in the bottom layer don't use those in the top layer at all. Even for a program this small, a little architecture helps make the code more manageable.

Second, I've tried as much as possible to decompose tasks into small functions so that each does one tiny, understandable task. Some of these functions are used by only one other function. I find this helpful because it lets me name each tiny subtask, allowing me to better express the intention of the code.

But before all the architecture, there's this:

```
(ns pegthing.core
  (require [clojure.set :as set])
  (:gen-class))

(declare successful-move prompt-move game-over query-rows)
```

I'll explain the functions here in more detail in Chapter 6. If you're curious about what's going on, the short explanation is that (`require [clojure.set :as set]`) allows you to easily use functions in the `clojure.set` namespace, (`:gen-class`) allows you to run the program from the command line, and (`declare successful-move prompt-move game-over query-rows`) allows functions to refer to those names before they're defined. If that doesn't quite make sense yet, trust that I will explain it soon.

Creating the Board

The data structure you use to represent the board should make it easy for you to print the board, check whether a player has made a valid move, actually perform a move, and check whether the game is over. You could structure the board in many ways to allow these tasks. In this case, you'll represent the board using a map with numerical keys corresponding to each board position and values containing information about that position's connections. The map will also contain a `:rows` key, storing the total number of rows. Figure 5-3 shows a board with each position numbered.

Figure 5-3: The numbered pegboard

Here's the data structure built to represent it:

```
{1  {:pegged true, :connections {6 3, 4 2}},
 2  {:pegged true, :connections {9 5, 7 4}},
 3  {:pegged true, :connections {10 6, 8 5}},
 4  {:pegged true, :connections {13 8, 11 7, 6 5, 1 2}},
 5  {:pegged true, :connections {14 9, 12 8}},
 6  {:pegged true, :connections {15 10, 13 9, 4 5, 1 3}},
 7  {:pegged true, :connections {9 8, 2 4}},
 8  {:pegged true, :connections {10 9, 3 5}},
 9  {:pegged true, :connections {7 8, 2 5}},
 10 {:pegged true, :connections {8 9, 3 6}},
 11 {:pegged true, :connections {13 12, 4 7}},
 12 {:pegged true, :connections {14 13, 5 8}},
 13 {:pegged true, :connections {15 14, 11 12, 6 9, 4 8}},
 14 {:pegged true, :connections {12 13, 5 9}},
 15 {:pegged true, :connections {13 14, 6 10}},
 :rows 5}
```

You might be wondering why, when you play the game, each position is represented by a letter but here the positions are represented by numbers. Using numbers for the internal representation allows you to take advantage of some mathematical properties of the board layout when validating and making moves. Letters, on the other hand, are better for display because they take up only one character space. Some conversion functions are covered in "Rendering and Printing the Board" on page 120.

; makes sense

In the data structure, each position is associated with a map that reads something like this:

```
{:pegged true, :connections {6 3, 4 2}}
```

The meaning of `:pegged` is clear; it represents whether that position has a peg in it. `:connections` is a bit more cryptic. It's a map where each key identifies a *legal destination*, and each value represents *the position that would be jumped over*. So pegs in position 1, for example, can jump *to* position 6 *over* position 3. This might seem backward, but you'll learn the rationale for it later when you see how move validation is implemented.

Now that you've seen what the final map representing the board should look like, we can start exploring the functions that actually build up this map in the program. You won't simply start assigning mutable states willy-nilly to represent each position and whether it's pegged or not. Instead, you'll use nested recursive function calls to build up the final board position by position. It's analogous to the way you created the glamour shot caption earlier, deriving new data from input by passing an argument through a chain of functions to get your final result.

The first few expressions in this section of the code deal with triangular numbers. Triangular numbers are generated by adding the first *n* natural numbers. The first triangular number is 1, the second is 3 (1 + 2), the third is 6 (1 + 2 + 3), and so on. These numbers line up nicely with the position numbers at the end of every row on the board, which will turn out to be a very useful property. First, you define the function `tri*`, which can create a lazy sequence of triangular numbers:

```
(defn tri*
  "Generates lazy sequence of triangular numbers"
  ([] (tri* 0 1))
  ([sum n]
     (let [new-sum (+ sum n)]
       (cons new-sum (lazy-seq (tri* new-sum (inc n)))))))
```

To quickly recap how this works, calling `tri*` with no arguments will recursively call `(tri* 0 1)`. This returns a lazy list whose element is `new-sum`, which in this case evaluates to `1`. The lazy list includes a recipe for generating the next element of the list when it's requested, as described in Chapter 4.

The next expression calls `tri*`, actually creating the lazy sequence and binding it to `tri`: ; this is what's confusing me right now. use defn to define, but def to… call the function?

```
(def tri (tri*))
```

You can verify that it actually works:

```
(take 5 tri)
; => (1 3 6 10 15)
```

And the next few functions operate on the sequence of triangular numbers. `triangular?` figures out if its argument is in the `tri` lazy sequence. It works by using `take-while` to create a sequence of triangular numbers whose last element is a triangular number that's less than or equal to the argument. Then it compares the last element to the argument:

```
(defn triangular?
  "Is the number triangular? e.g. 1, 3, 6, 10, 15, etc"
  [n]
  (= n (last (take-while #(>= n %) tri))))
(triangular? 5)
; => false

(triangular? 6)
; => true
```

Next, there's `row-tri`, which takes a row number and gives you the triangular number at the end of that row:

```
(defn row-tri
  "The triangular number at the end of row n"
  [n]
  (last (take n tri)))
(row-tri 1)
; => 1

(row-tri 2)
; => 3

(row-tri 3)
; => 6
```

Lastly, there's `row-num`, which takes a board position and returns the row that it belongs to:

```
(defn row-num
  "Returns row number the position belongs to: pos 1 in row 1,
  positions 2 and 3 in row 2, etc"
  [pos]
  (inc (count (take-while #(> pos %) tri))))
(row-num 1)
; => 1
```

```
(row-num 5)
; => 3
```

After that comes `connect`, which is used to actually form a mutual connection between two positions:

```
(defn connect
  "Form a mutual connection between two positions"
  [board max-pos pos neighbor destination]
  (if (<= destination max-pos)
    (reduce (fn [new-board [p1 p2]]
              (assoc-in new-board [p1 :connections p2] neighbor))
            board
            [[pos destination] [destination pos]])
    board))

(connect {} 15 1 2 4)
; => {1 {:connections {4 2}}
      4 {:connections {1 2}}}
```

The first thing `connect` does is check whether the destination is actually a position on the board by confirming that it's less than the board's max position. For example, if you have five rows, the max position is 15. However, when the game board is created, the `connect` function will be called with arguments like `(connect {} 15 7 16 22)`. The `if` statement at the beginning of `connect` makes sure your program doesn't allow such outrageous connections, and simply returns the unmodified board when you ask it to do something silly.

Next, `connect` uses recursion through `reduce` to progressively build up the final state of the board. In this example, you're reducing over the nested vectors `[[1 4] [4 1]]`. This is what allows you to return an updated board with both `pos` and `destination` (1 and 4) pointing to each other in their connections.

The anonymous function passed to `reduce` uses a function, `assoc-in`, which you haven't seen before. Whereas the function `get-in` lets you look up values in nested maps, `assoc-in` lets you return a new map with the given value at the specified nesting. Here are a couple of examples:

```
(assoc-in {} [:cookie :monster :vocals] "Finntroll")
; => {:cookie {:monster {:vocals "Finntroll"}}}

(get-in {:cookie {:monster {:vocals "Finntroll"}}} [:cookie :monster])
; => {:vocals "Finntroll"}

(assoc-in {} [1 :connections 4] 2)
; => {1 {:connections {4 2}}}
```

In these examples, you can see that new, nested maps are created to accommodate all the keys provided.

Now we have a way to connect two positions, but how should the program choose two positions to connect in the first place? That's handled by `connect-right`, `connect-down-left`, and `connect-down-right`:

```
(defn connect-right
  [board max-pos pos]
  (let [neighbor (inc pos)
        destination (inc neighbor)]
    (if-not (or (triangular? neighbor) (triangular? pos))
      (connect board max-pos pos neighbor destination)
      board)))

(defn connect-down-left
  [board max-pos pos]
  (let [row (row-num pos)
        neighbor (+ row pos)
        destination (+ 1 row neighbor)]
    (connect board max-pos pos neighbor destination)))

(defn connect-down-right
  [board max-pos pos]
  (let [row (row-num pos)
        neighbor (+ 1 row pos)
        destination (+ 2 row neighbor)]
    (connect board max-pos pos neighbor destination)))
```

These functions each take the board's max position and a board position and use a little triangle math to figure out which numbers to feed to `connect`. For example, `connect-down-left` will attempt to connect position 1 to position 4. In case you're wondering why the functions `connect-left`, `connect-up-left`, and `connect-up-right` aren't defined, the reason is that the existing functions actually cover these cases. `connect` returns a board with the mutual connection established; when 4 *connects right* to 6, 6 *connects left* to 4. Here are a couple of examples:

```
(connect-down-left {} 15 1)
; => {1 {:connections {4 2}
      4 {:connections {1 2}}}}

(connect-down-right {} 15 3)
; => {3  {:connections {10 6}}
      10 {:connections {3 6}}}
```

In the first example, `connect-down-left` takes an empty board with a max position of 15 and returns a new board populated with the mutual connection between the 1 position and the position below and to the left of it. `connect-down-right` does something similar, returning a board populated with the mutual connection between 3 and the position below it and to its right.

The next function, `add-pos`, is interesting because it actually reduces on a vector of *functions*, applying each in turn to build up the resulting board. But first it updates the board to indicate that a peg is in the given position:

```
(defn add-pos
  "Pegs the position and performs connections"
  [board max-pos pos]
  (let [pegged-board (assoc-in board [pos :pegged] true)]
    (reduce (fn [new-board connection-creation-fn]
              (connection-creation-fn new-board max-pos pos))
            pegged-board
            [connect-right connect-down-left connect-down-right])))

(add-pos {} 15 1)
{1 {:connections {6 3, 4 2}, :pegged true}
 4 {:connections {1 2}}
 6 {:connections {1 3}}}
```

;; that's pretty cool. I bet it would be easy to do the same in Javascript, with a rewritten "reduce-fns" function

function reduce_fns(…fns) {

} // started it for ya…

It's like this function is first saying, in the `pegged-board` binding, "Add a peg to the board at position X." Then, in `reduce`, it says, "Take the board with its new peg at position X, and try to connect position X to a legal, rightward position. Take the board that results from that operation, and try to connect position X to a legal, down-left position. Finally, take the board that results from *that* operation, and try to connect position X to a legal, down-right position. Return the resulting board."

Reducing over functions like this is another way of composing functions. To illustrate, here's another way of defining the `clean` function in Listing 5-1 (page 103):

```
(defn clean
  [text]
  (reduce (fn [string string-fn] (string-fn string))
          text
          [s/trim #(s/replace % #"lol" "LOL")]))
```

This redefinition of `clean` reduces a vector of functions by applying the first function, `s/trim`, to an initial string, then applying the next function, the anonymous function `#(s/replace % #"lol" "LOL")`, to the result.

Reducing over a collection of functions is not a technique you'll use often, but it's occasionally useful, and it demonstrates the versatility of functional programming.

Last among our board creation functions is `new-board`:

```
(defn new-board
  "Creates a new board with the given number of rows"
  [rows]
  (let [initial-board {:rows rows}
        max-pos (row-tri rows)]
    (reduce (fn [board pos] (add-pos board max-pos pos))
            initial-board
            (range 1 (inc max-pos)))))
```

The code first creates the initial, empty board and gets the max position. Assuming that you're using five rows, the max position will be 15. Next, the function uses `(range 1 (inc max-pos))` to get a list of numbers from 1 to 15, otherwise known as the board's positions. Finally, it reduces over the list of positions. Each iteration of the reduction calls `(add-pos board max-pos pos)`, which, as you saw earlier, takes an existing board and returns a new one with the position added.

Moving Pegs

The next section of code validates and performs peg moves. Many of the functions (`pegged?`, `remove-peg`, `place-peg`, `move-peg`) are simple, self-explanatory one-liners:

```
(defn pegged?
  "Does the position have a peg in it?"
  [board pos]
  (get-in board [pos :pegged]))

(defn remove-peg
  "Take the peg at given position out of the board"
  [board pos]
  (assoc-in board [pos :pegged] false))

(defn place-peg
  "Put a peg in the board at given position"
  [board pos]
  (assoc-in board [pos :pegged] true))

(defn move-peg
  "Take peg out of p1 and place it in p2"
  [board p1 p2]
  (place-peg (remove-peg board p1) p2))
```

Let's take a moment to appreciate how neat this code is. This is where you would usually perform mutation in an object-oriented program; after all, how else would you change the board? However, these are all pure functions, and they do the job admirably. I also like that you don't need the overhead of classes to use these little guys. It feels somehow lighter to program like this.

Next up is `valid-moves`:

```
(defn valid-moves
  "Return a map of all valid moves for pos, where the key is the
  destination and the value is the jumped position"
  [board pos]
  (into {}
        (filter (fn [[destination jumped]]
                  (and (not (pegged? board destination))
                       (pegged? board jumped)))
                (get-in board [pos :connections]))))
```

This code goes through each of the given position's connections and tests whether the destination position is empty and the jumped position has a peg. To see this in action, you can create a board with the 4 position empty:

```
(def my-board (assoc-in (new-board 5) [4 :pegged] false))
```

Figure 5-4 shows what that board would look like.

Figure 5-4: A pegboard with the 4 position empty

Given this board, positions 1, 6, and 11 have valid moves, but all others don't:

;; errata: position 13 also has a valid move

```
(valid-moves my-board 1)  ; => {4 2}
(valid-moves my-board 6)  ; => {4 5}
(valid-moves my-board 11) ; => {4 7}
(valid-moves my-board 5)  ; => {}
(valid-moves my-board 8)  ; => {}
```

You might be wondering why `valid-moves` returns a map instead of, say, a set or vector. The reason is that returning a map allows you to easily look up a destination position to check whether a specific move is valid, which is what `valid-move?` (the next function) does:

```
(defn valid-move?
  "Return jumped position if the move from p1 to p2 is valid, nil
  otherwise"
  [board p1 p2]
  (get (valid-moves board p1) p2))

(valid-move? my-board 8 4) ; => nil
(valid-move? my-board 1 4) ; => 2
```

Notice that `valid-move?` looks up the destination position from the map and then returns the position of the peg that would be jumped over. This is another nice benefit of having `valid-moves` return a map, because the

jumped position retrieved from the map is exactly what we want to pass on to the next function, `make-move`. When you take the time to construct a rich data structure, it's easier to perform useful operations.

```
(defn make-move
  "Move peg from p1 to p2, removing jumped peg"
  [board p1 p2]  ;; this syntax is kinda weird
  (if-let [jumped (valid-move? board p1 p2)]
    (move-peg (remove-peg board jumped) p1 p2)))
```

`if-let` is a nifty way to say, "If an expression evaluates to a truthy value, then bind that value to a name the same way that I can in a `let` expression. Otherwise, if I've provided an `else` clause, perform that `else` clause; if I haven't provided an `else` clause, return `nil`." In this case, the test expression is `(valid-move? board p1 p2)`, and you're assigning the result to the name `jumped` if the result is truthy. That's used in the call to `move-peg`, which returns a new board. You don't supply an `else` clause, so if the move isn't valid, the return value of the whole expression is `nil`.

Finally, the function `can-move?` is used to determine whether the game is over by finding the first pegged positions with moves available:

```
(defn can-move?
  "Do any of the pegged positions have valid moves?"
  [board]
  (some (comp not-empty (partial valid-moves board))
        (map first (filter #(get (second %) :pegged) board))))
```

;; so map is taking FIRST as the function, and we're using FILTER to pass in the collection that we want mapped over

The question mark at the end of this function name indicates it's a *predicate function*, a function that's meant to be used in Boolean expressions. *Predicate* is taken from predicate logic, which concerns itself with determining whether a statement is true or false. (You've already seen some built-in predicate functions, like `empty?` and `every?`.)

`can-move?` works by getting a sequence of all pegged positions with `(map first (filter #(get (second %) :pegged) board))`. You can break this down further into the `filter` and `map` function calls: because `filter` is a seq function, it converts `board`, a map, into a seq of two-element vectors (also called *tuples*), which looks something like this: ;; ah, so that's why we're using SECOND

```
([1 {:connections {6 3, 4 2}, :pegged true}]
 [2 {:connections {9 5, 7 4}, :pegged true}])
```

The first element of the tuple is a position number, and the second is that position's information. `filter` then applies the anonymous function `#(get (second %) :pegged)` to each of these tuples, filtering out the tuples where the position's information indicates that the position is not currently housing a peg. Finally, the result is passed to `map`, which calls `first` on each tuple to grab just the position number from the tuples.

;; FIRST and SECOND seem like a really simple, natural way to work with data structures (esp. tuples)

After you get a seq of pegged positions numbers, you call a predicate function on each one to find the first position that returns a truthy value. The

predicate function is created with `(comp not-empty (partial valid-moves board))`. The idea is to first return a map of all valid moves for a position and then test whether that map is empty.

First, the expression `(partial valid-moves board)` derives an anonymous function from `valid-moves` with the first argument, `board`, filled in using `partial` (because you're using the same board each time you call `valid-moves`). The new function can take a position and return the map of all its valid moves for the current board. ;; ahh, now it makes sense

Second, you use `comp` to compose this function with `not-empty`. This function is self-descriptive; it returns `true` if the given collection is empty and `false` otherwise.

What's most interesting about this bit of code is that you're using a chain of functions to derive a new function, similar to how you use chains of functions to derive new data. In Chapter 3, you learned that Clojure treats functions as data in that functions can receive functions as arguments and return them. Hopefully, this shows why that feature is fun and useful.

Rendering and Printing the Board

The first few expressions in the board representation and printing section just define constants:

```
(def alpha-start 97)
(def alpha-end 123)
(def letters (map (comp str char) (range alpha-start alpha-end)))
(def pos-chars 3)
```

The bindings `alpha-start` and `alpha-end` set up the beginning and end of the numerical values for the letters *a* through *z*. We use those to build up a seq of `letters`. `char`, when applied to an integer, returns the character corresponding to that integer, and `str` turns the `char` into a string. `pos-chars` is used by the function `row-padding` to determine how much spacing to add to the beginning of each row. The next few definitions, `ansi-styles`, `ansi`, and `colorize` output colored text to the terminal.

The functions `render-pos`, `row-positions`, `row-padding`, and `render-row` create strings to represent the board:

```
(defn render-pos
  [board pos]
  (str (nth letters (dec pos))
       (if (get-in board [pos :pegged])
         (colorize "O" :blue)
         (colorize "-" :red))))

(defn row-positions
  "Return all positions in the given row"
  [row-num]
  (range (inc (or (row-tri (dec row-num)) 0))
         (inc (row-tri row-num))))
```

```
(defn row-padding
  "String of spaces to add to the beginning of a row to center it"
  [row-num rows]
  (let [pad-length (/ (* (- rows row-num) pos-chars) 2)]
    (apply str (take pad-length (repeat " "))))

(defn render-row
  [board row-num]
  (str (row-padding row-num (:rows board))
       (clojure.string/join " " (map (partial render-pos board)
                                     (row-positions row-num)))))
```

If you work from the bottom up, you can see that `render-row` calls each of the functions above it to return the string representation of the given row. Notice the expression `(map (partial render-pos board) (row-positions row-num))`. This demonstrates a good use case for partials by applying the same function multiple times with one or more arguments filled in, just like in the `can-move?` function shown earlier.

Notice too that `render-pos` uses a letter rather than a number to identify each position. This saves a little space when the board is displayed, because it allows only one character per position to represent a five-row board.

Finally, `print-board` merely iterates over each row number with `doseq`, printing the string representation of that row:

```
(defn print-board
  [board]
  (doseq [row-num (range 1 (inc (:rows board)))]
    (println (render-row board row-num))))
```

You use `doseq` when you want to perform side-effecting operations (like printing to a terminal) on the elements of a collection. The vector that immediately follows the name `doseq` describes how to bind all the elements in a collection to a name one at a time so you can operate on them. In this instance, you're assigning the numbers 1 through 5 (assuming there are five rows) to the name `row-num` so you can print each row.

Although printing the board technically falls under *interaction*, I wanted to show it here with the rendering functions. When I first started writing this game, the `print-board` function also generated the board's string representation. However, now `print-board` defers all rendering to pure functions, which makes the code easier to understand and decreases the surface area of our impure functions. ;; I like the way he put that

Player Interaction

The next collection of functions handles player interaction. First, there's `letter->pos`, which converts a letter (which is how the positions are displayed and identified by players) to the corresponding position number:

```
(defn letter->pos
  "Converts a letter string to the corresponding position number"
```

```
  [letter]
  (inc (- (int (first letter)) alpha-start)))
```

Next, the helper function `get-input` allows you to read and clean the player's input. You can also provide a default value, which is used if the player presses ENTER without typing anything:

```
(defn get-input
  "Waits for user to enter text and hit enter, then cleans the input"
  ([] (get-input nil))
  ([default]
     (let [input (clojure.string/trim (read-line))]
       (if (empty? input)
         default
         (clojure.string/lower-case input)))))
```

The next function, `characters-as-strings`, is a tiny helper function used by `prompt-move` to take in a string and return a collection of letters with all nonalphabetic input discarded:

```
(characters-as-strings "a   b")
; => ("a" "b")

(characters-as-strings "a   cb")
; => ("a" "c" "b")
```

Next, `prompt-move` reads the player's input and acts on it:

```
(defn prompt-move
  [board]
  (println "\nHere's your board:")
  (print-board board)
  (println "Move from where to where? Enter two letters:")
  (let [input (map letter->pos (characters-as-strings (get-input)))]
    (if-let [new-board (make-move❶ board (first input) (second input))]
      (user-entered-valid-move new-board)
      (user-entered-invalid-move board))))
```

At ❶, `make-move` returns `nil` if the player's move was invalid, and you use that information to inform her of her mistake with the `user-entered-invalid-move` function. You pass the unmodified board to `user-entered-invalid-move` so that it can prompt the player with the board again. Here's the function definition:

```
(defn user-entered-invalid-move
  "Handles the next step after a user has entered an invalid move"
  [board]
  (println "\n!!! That was an invalid move :(\n")
  (prompt-move board))
```

However, if the move is valid, the `new-board` is passed off to `user-entered-valid-move`, which hands control back to `prompt-move` if there are still moves to be made:

```
(defn user-entered-valid-move
  "Handles the next step after a user has entered a valid move"
  [board]
  (if (can-move? board)
    (prompt-move board)
    (game-over board)))
```

In our board creation functions, we saw how recursion was used to build up a value using immutable data structures. The same thing is happening here, only it involves two mutually recursive functions and some user input. No mutable attributes in sight!

What happens when the game is over? This is what happens:

```
(defn game-over
  "Announce the game is over and prompt to play again"
  [board]
  (let [remaining-pegs (count (filter :pegged (vals board)))]
    (println "Game over! You had" remaining-pegs "pegs left:")
    (print-board board)
    (println "Play again? y/n [y]")
    (let [input (get-input "y")]
      (if (= "y" input)
        (prompt-rows)
        (do
          (println "Bye!")
          (System/exit 0))))))
```

All that's going on here is that the game tells you how you did, prints the final board, and prompts you to play again. If you select *y*, the game calls `prompt-rows`, which brings us to the final set of functions, those used to start a new game:

```
(defn prompt-empty-peg
  [board]
  (println "Here's your board:")
  (print-board board)
  (println "Remove which peg? [e]")
  (prompt-move (remove-peg board (letter->pos (get-input "e")))))

(defn prompt-rows
  []
  (println "How many rows? [5]")
  (let [rows (Integer. (get-input 5))
        board (new-board rows)]
    (prompt-empty-peg board)))
```

You use `prompt-rows` to start a game, getting the player's input on how many rows to include. Then you pass control on to `prompt-empty-peg` so the player can tell the game which peg to remove first. From there, the program prompts you for moves until there aren't any moves left.

Even though all of this program's side effects are relatively harmless (all you're doing is prompting and printing), sequestering them in their own functions like this is a best practice for functional programming. In general, you will reap more benefits from functional programming if you identify the bits of functionality that are referentially transparent and side-effect free, and place those bits in their own functions. These functions are not capable of causing bizarre bugs in unrelated parts of your program. They're easier to test and develop in the REPL because they rely only on the arguments you pass them, not on some complicated hidden state object.

Summary

Pure functions are referentially transparent and side-effect free, which makes them easy to reason about. To get the most from Clojure, try to keep your impure functions to a minimum. In an immutable world, you use recursion instead of `for`/`while` loops, and function composition instead of successions of mutations. Pure functions allow powerful techniques like function composition functions and memoization. They're also super fun!

Exercises

One of the best ways to develop your functional programming skills is to try to implement existing functions. To that end, most of the following exercises suggest a function for you to implement, but don't stop there; go through the Clojure cheat sheet (*http://clojure.org/cheatsheet/*) and pick more!

;; https://clojure.org/api/cheatsheet

1. You used `(comp :intelligence :attributes)` to create a function that returns a character's intelligence. Create a new function, `attr`, that you can call like `(attr :intelligence)` and that does the same thing.
2. Implement the `comp` function.
3. Implement the `assoc-in` function. Hint: use the `assoc` function and define its parameters as `[m [k & ks] v]`.
4. Look up and use the `update-in` function.
5. Implement `update-in`.

6

ORGANIZING YOUR PROJECT: A LIBRARIAN'S TALE

Within each of us lives a librarian named Melvil, a fantastical creature who delights in the organizational arts. Day and night, Melvil yearns to bring order to your codebase. Fortunately, Clojure provides a suite of tools designed specifically to aid this homunculus in its constant struggle against the forces of chaos.

These tools help you organize your code by grouping together related functions and data. They also prevent name collisions so you don't accidentally overwrite someone else's code or vice versa. Join me in a tale of suspense and mystery as you learn how to use these tools and solve the heist of a lifetime! By the end of the saga, you'll understand the following:

- What `def` does
- What namespaces are and how to use them
- The relationship between namespaces and the filesystem

- How to use `refer`, `alias`, `require`, `use`, and `ns`
- How to organize Clojure projects using the filesystem

I'll start with a high-level overview of Clojure's organizational system, which works much like a library. Melvil quivers with excitement!

Your Project as a Library

Real-world libraries store collections of objects, such as books, magazines, and DVDs. They use addressing systems, so when you're given an object's address, you can navigate to the physical space and retrieve the object.

Of course, no human being would be expected to know offhand what a book's or DVD's address is. That's why libraries record the association between an object's title and its address and provide tools for searching these records. In ye olden times before computers, libraries provided card catalogs, which were cabinets filled with paper cards containing each book's title, author, "address" (its Dewey decimal or Library of Congress number), and other info.

For example, to find *The Da Vinci Code*, you would riffle through the title catalog (cards ordered by title) until you found the correct card. On that card you would see the address *813.54* (if it's using the Dewey decimal system), navigate your library to find the shelf where *The Da Vinci Code* resides, and engage in the literary and/or hate-reading adventure of your lifetime.

It's useful to imagine a similar setup in Clojure. I think of Clojure as storing objects (like data structures and functions) in a vast set of numbered shelves. No human being could know offhand which shelf an object is stored in. Instead, we give Clojure an identifier that it uses to retrieve the object.

For this to be successful, Clojure must maintain the associations between our identifiers and shelf addresses. It does this by using *namespaces*. Namespaces contain maps between human-friendly *symbols* and references to shelf addresses, known as *vars*, much like a card catalog.

Technically, namespaces are objects of type `clojure.lang.Namespace`, and you can interact with them just like you can with Clojure data structures. For example, you can refer to the current namespace with `*ns*`, and you can get its name with `(ns-name *ns*)`:

```
(ns-name *ns*)
; => user
```

When you start the REPL, for example, you're in the `user` namespace (as you can see here). The prompt shows the current namespace using something like `user=>`.

The idea of a current namespace implies that you can have more than one, and indeed Clojure allows you to create as many namespaces as you want (although technically, there might be an upper limit to the number of names you can create). In Clojure programs, you are always *in* a namespace.

As for symbols, you've been using them this entire time without even realizing it. For example, when you write `(map inc [1 2])`, both `map` and `inc` are symbols. Symbols are data types within Clojure, and I'll explain them thoroughly in the next chapter. For now, all you need to know is that when you give Clojure a symbol like `map`, it finds the corresponding var in the current namespace, gets a shelf address, and retrieves an object from that shelf for you—in this case, the function that `map` refers to. If you want to just use the symbol itself, and not the thing it refers to, you have to quote it. Quoting any Clojure form tells Clojure not to evaluate it but to treat it as data. The next few examples show what happens when you quote a form.

```
❶ inc
  ; => #<core$inc clojure.core$inc@30132014>

❷ 'inc
  ; => inc

❸ (map inc [1 2])
  ; => (2 3)

❹ '(map inc [1 2])
  ; => (map inc [1 2])
```

When you evaluate `inc` in the REPL at ❶, it prints out the textual representation of the function that `inc` refers to. Next, you quote `inc` at ❷, so the result is the symbol `inc`. Then, you evaluate a familiar `map` application at ❸ and get a familiar result. After that, you quote the entire list data structure at ❹, resulting in an unevaluated list that includes the `map` symbol, the `inc` symbol, and a vector.

Now that you know about Clojure's organization system, let's look at how to use it.

Storing Objects with def

The primary tool in Clojure for storing objects is `def`. Other tools like `defn` use `def` under the hood. Here's an example of def in action:

```
(def great-books ["East of Eden" "The Glass Bead Game"])
; => #'user/great-books

great-books
; => ["East of Eden" "The Glass Bead Game"]
```

This code tells Clojure:

1. Update the current namespace's map with the association between `great-books` and the var.
2. Find a free storage shelf.
3. Store `["East of Eden" "The Glass Bead Game"]` on the shelf.

4. Write the address of the shelf on the var.
5. Return the var (in this case, `#'user/great-books`).

This process is called *interning* a var. You can interact with a namespace's map of symbols-to-interned-vars using `ns-interns`. Here's how you'd get a map of interned vars:

```
(ns-interns *ns*)
; => {great-books #'user/great-books}
```

You can use the `get` function to get a specific var:

```
(get (ns-interns *ns*) 'great-books)
; => #'user/great-books
```

By evaluating `(ns-map *ns*)`, you can also get the full map that the namespace uses for looking up a var when given a symbol. `(ns-map *ns*)` gives you a very large map that I won't print here, but try it out!

`#'user/great-books` is the *reader form* of a var. I'll explain more about reader forms in Chapter 7. For now, just know that you can use `#'` to grab hold of the var corresponding to the symbol that follows; `#'user/great-books` lets you use the var associated with the symbol `great-books` within the `user` namespace. We can `deref` vars to get the objects they point to:

```
(deref #'user/great-books)
; => ["East of Eden" "The Glass Bead Game"]
```

This is like telling Clojure, "Get the shelf number from the var, go to that shelf number, grab what's on it, and give it to me!"

But normally, you would just use the symbol:

```
great-books
; => ["East of Eden" "The Glass Bead Game"]
```

This is like telling Clojure, "Retrieve the var associated with great-books and deref that bad Jackson."

So far so good, right? Well, brace yourself, because this idyllic paradise of organization is about to be turned upside down! Call `def` again with the same symbol:

```
(def great-books ["The Power of Bees" "Journey to Upstairs"])
great-books
; => ["The Power of Bees" "Journey to Upstairs"]
```

The var has been updated with the address of the new vector. It's like you used white-out on the address on a card in the card catalog and then wrote a new address. The result is that you can no longer ask Clojure to find the first vector. This is referred to as a *name collision*. Chaos! Anarchy!

You may have experienced this in other programming languages. JavaScript is notorious for it, and it happens in Ruby as well. It's a problem because you can unintentionally overwrite your own code, and you also have no guarantee that a third-party library won't overwrite your code. Melvil recoils in horror! Fortunately, Clojure allows you to create as many namespaces as you like so you can avoid these collisions.

Creating and Switching to Namespaces

Clojure has three tools for creating namespaces: the function `create-ns`, the function `in-ns`, and the macro `ns`. You'll mostly use the `ns` macro in your Clojure files, but I'll hold off on explaining it for a few pages because it combines many tools, and it's easier to understand after I discuss each of the other tools.

`create-ns` takes a symbol, creates a namespace with that name if it doesn't exist already, and returns the namespace:

```
user=> (create-ns 'cheese.taxonomy)
; => #<Namespace cheese.taxonomy>
```

You can use the returned namespace as an argument in a function call:

```
user=> (ns-name (create-ns 'cheese.taxonomy))
; => cheese-taxonomy
```

In practice, you'll probably never use `create-ns` in your code, because it's not very useful to create a namespace and not move into it. Using `in-ns` is more common because it creates the namespace if it doesn't exist *and* switches to it, as shown in Listing 6-1.

```
user=> (in-ns 'cheese.analysis)
; => #<Namespace cheese.analysis>
```

Listing 6-1: Using `in-ns` to create a namespace and switch to it

Notice that your REPL prompt is now `cheese.analysis>`, indicating that you are indeed in the new namespace you just created. Now when you use `def`, it will store the named object in the `cheese.analysis` namespace.

But what if you want to use functions and data from other namespaces? To do that, you can use a *fully qualified* symbol. The general form is *`namespace/name`*:

```
cheese.analysis=> (in-ns 'cheese.taxonomy)
cheese.taxonomy=> (def cheddars ["mild" "medium" "strong" "sharp" "extra sharp"])
cheese.taxonomy=> (in-ns 'cheese.analysis)
```

```
cheese.analysis=> cheddars
; => Exception: Unable to resolve symbol: cheddars in this context
```

This creates a new namespace, `cheese.taxonomy`, defines `cheddars` in that namespace, and then switches back to the `cheese.analysis` namespace. You'll get an exception if you try to refer to the `cheese.taxonomy` namespace's `cheddars` from within `cheese.analysis`, but using the fully qualified symbol works:

```
cheese.analysis=> cheese.taxonomy/cheddars
; => ["mild" "medium" "strong" "sharp" "extra sharp"]
```

Typing these fully qualified symbols can quickly become a nuisance. For instance, say I'm an extremely impatient academic specializing in semiotics-au-fromage, or the study of symbols as they relate to cheese.

Suddenly, the worst conceivable thing that could possibly happen happens! All across the world, sacred and historically important cheeses have gone missing. Wisconsin's Standard Cheddar: gone! The Great Cheese Jars of Tutankhamun: stolen! The Cheese of Turin: replaced with a hoax cheese! This threatens to throw the world into total chaos for some reason! Naturally, as a distinguished cheese researcher, I am honor-bound to solve this mystery. Meanwhile, I'm being chased by the Illuminati, the Freemasons, and the Foot Clan!

;; I lol'ed

Because I'm an academic, I attempt to solve this mystery the best way I know how—by heading to the library and researching the shit out of it. My trusty assistant, Clojure, accompanies me. As we bustle from namespace to namespace, I shout at Clojure to hand me one thing after another.

But Clojure is kind of dumb and has a hard time figuring out what I'm referring to. From within the `user` namespace, I belt out, "`join`! Give me `join`!"—specks of spittle flying from my mouth. "`RuntimeException: Unable to resolve symbol: join`," Clojure whines in response. "For the love of brie, just hand me `clojure.string/join`!" I retort, and Clojure dutifully hands me the function I was looking for.

My voice gets hoarse. I need some way to tell Clojure what objects to get me without having to use the fully qualified symbol every. damn. time.

Luckily, Clojure provides the `refer` and `alias` tools that let me yell at it more succinctly.

refer

`refer` gives you fine-grained control over how you refer to objects in other namespaces. Fire up a new REPL session and try the following. Keep in mind that it's okay to play around with namespaces like this in the REPL, but you don't want your Clojure files to look like this; the proper way to structure your files is covered in "Real Project Organization" on page 133.

```
user=> (in-ns 'cheese.taxonomy)
cheese.taxonomy=> (def cheddars ["mild" "medium" "strong" "sharp" "extra sharp"])
cheese.taxonomy=> (def bries ["Wisconsin" "Somerset" "Brie de Meaux" "Brie de Melun"])
cheese.taxonomy=> (in-ns 'cheese.analysis)
```

```
cheese.analysis=> (clojure.core/refer 'cheese.taxonomy)
cheese.analysis=> bries
; => ["Wisconsin" "Somerset" "Brie de Meaux" "Brie de Melun"]

cheese.analysis=> cheddars
; => ["mild" "medium" "strong" "sharp" "extra sharp"]
```

This code creates a `cheese.taxonomy` namespace and two vectors within it: `cheddars` and `bries`. Then it creates and moves to a new namespace called `cheese.analysis`. Calling `refer` with a namespace symbol lets you refer to the corresponding namespace's objects without having to use fully qualified symbols. It does this by updating the current namespace's symbol/object map. You can see the new entries like this:

```
cheese.analysis=> (clojure.core/get (clojure.core/ns-map clojure.core/*ns*) 'bries)
; => #'cheese.taxonomy/bries

cheese.analysis=> (clojure.core/get (clojure.core/ns-map clojure.core/*ns*) 'cheddars)
; => #'cheese.taxonomy/cheddars
```

It's as if Clojure

1. Calls `ns-interns` on the `cheese.taxonomy` namespace
2. Merges that with the `ns-map` of the current namespace
3. Makes the result the new `ns-map` of the current namespace

When you call `refer`, you can also pass it the filters `:only`, `:exclude`, and `:rename`. As the names imply, `:only` and `:exclude` restrict which symbol/var mappings get merged into the current namespace's `ns-map`. `:rename` lets you use different symbols for the vars being merged in. Here's what would happen if we had modified the preceding example to use `:only`:

```
cheese.analysis=> (clojure.core/refer 'cheese.taxonomy :only ['bries])
cheese.analysis=> bries
; => ["Wisconsin" "Somerset" "Brie de Meaux" "Brie de Melun"]
cheese.analysis=> cheddars
; => RuntimeException: Unable to resolve symbol: cheddars
```

And here's `:exclude` in action:

```
cheese.analysis=> (clojure.core/refer 'cheese.taxonomy :exclude ['bries])
cheese.analysis=> bries
; => RuntimeException: Unable to resolve symbol: bries
cheese.analysis=> cheddars
; => ["mild" "medium" "strong" "sharp" "extra sharp"]
```

Lastly, a `:rename` example:

```
cheese.analysis=> (clojure.core/refer 'cheese.taxonomy :rename {'bries 'yummy-bries})
cheese.analysis=> bries
; => RuntimeException: Unable to resolve symbol: bries
```

```
cheese.analysis=> yummy-bries
; => ["Wisconsin" "Somerset" "Brie de Meaux" "Brie de Melun"]
```

Notice that in these last examples we have to use the fully qualified names of all the objects in `clojure.core`, like `clojure.core/ns-map` and `clojure.core/refer`. We didn't have to do that in the `user` namespace. That's because the REPL automatically refers `clojure.core` within the `user` namespace. You can make your life easier by evaluating `(clojure.core/refer-clojure)` when you create a new namespace; this will refer the clojure.core namespace, and I'll be using it from now on. Instead of seeing `clojure.core/refer` in the examples, you'll only see `refer`.

Another thing to notice is that you have complete freedom over how you organize your functions and data across namespaces. This lets you sensibly group related functions and data together in the same namespace.

Sometimes you may want a function to be available only to other functions within the same namespace. Clojure allows you to define *private* functions using `defn-`:

```
(in-ns 'cheese.analysis)
;; Notice the dash after "defn"
(defn- private-function
  "Just an example function that does nothing"
  [])
```

If you try to call this function from another namespace or refer it, Clojure will throw an exception. You can see this when you evaluate the code at ❶ and ❷:

```
  cheese.analysis=> (in-ns 'cheese.taxonomy)
  cheese.taxonomy=> (clojure.core/refer-clojure)
❶ cheese.taxonomy=> (cheese.analysis/private-function)
❷ cheese.taxonomy=> (refer 'cheese.analysis :only ['private-function])
```

As you can see, even if you explicitly `refer` the function, you can't use the function from another namespace, because you made it private. (If you want to be tricky, you can still access the private var using the arcane syntax `@#'some/private-var`, but you'll rarely want to do that.)

alias

Compared to `refer`, `alias` is relatively simple. All it does is let you shorten a namespace name for using fully qualified symbols:

```
cheese.analysis=> (clojure.core/alias 'taxonomy 'cheese.taxonomy)
cheese.analysis=> taxonomy/bries
; => ["Wisconsin" "Somerset" "Brie de Meaux" "Brie de Melun"]
```

This code lets us use call symbols from the `cheese.taxonomy` namespace with the shorter alias `taxonomy`.

`refer` and `alias` are your two basic tools for referring to objects outside your current namespace! They're great aids to REPL development.

However, it's unlikely that you'd create your entire program in the REPL. In the next section, I'll cover everything you need to know to organize a real project with source code living on the filesystem.

Real Project Organization

Now that I've covered the building blocks of Clojure's organization system, I'll show you how to use them in real projects. I'll discuss the relationship between file paths and namespace names, explain how to load a file with `require` and `use`, and show how to use `ns` to set up a namespace.

The Relationship Between File Paths and Namespace Names

To kill two birds with one stone (or feed two birds with one seed, depending on how much of a hippie you are), I'll cover more on namespaces while we work on catching the pesky international cheese thief by mapping the locations of his heists. Run the following:

```
lein new app the-divine-cheese-code
```

This should create a directory structure that looks like this:

```
| .gitignore
| doc
| | intro.md
| project.clj
| README.md
| resources
| src
| | the_divine_cheese_code
| | | core.clj
| test
| | the_divine_cheese_code
| | | core_test.clj
```

Now, open *src/the_divine_cheese_code/core.clj*. You should see this on the first line:

```
(ns the-divine-cheese-code.core
  (:gen-class))
```

`ns` is the primary way to create and manage namespaces within Clojure. I'll explain it in full shortly. For now, though, just know that this line is very similar to the `in-ns` function we used in Listing 6-1. It creates a namespace if it doesn't exist and then switches to it. I also cover `(:gen-class)` in more detail in Chapter 12.

The name of the namespace is `the-divine-cheese-code.core`. In Clojure, there's a one-to-one mapping between a namespace name and the path of the file where the namespace is declared, according to the following conventions:

- When you create a directory with `lein` (as you did here), the source code's root is *src* by default.
- Dashes in namespace names correspond to underscores in the filesystem. So `the-divine-cheese-code` is mapped to *the_divine_cheese_code* on the filesystem.
- The component preceding a period (`.`) in a namespace name corresponds to a directory. For example, since `the-divine-cheese-code.core` is the namespace name, *the_divine_cheese_code* is a directory.
- The final component of a namespace corresponds to a file with the *.clj* extension; `core` is mapped to *core.clj*.

Your project will have one more namespace, `the-divine-cheese-code.visualization.svg`. Go ahead and create the file for it now:

```
mkdir src/the_divine_cheese_code/visualization
touch src/the_divine_cheese_code/visualization/svg.clj
```

Notice that the filesystem path follows these conventions. With the relationship between namespaces and the filesystem down, let's look at `require` and `use`.

Requiring and Using Namespaces

The code in the `the-divine-cheese-code.core` namespace will use the functions in the namespace `the-divine-cheese-code.visualization.svg` to create SVG markup. To use `svg`'s functions, `core` will have to *require* it. But first, let's add some code to *svg.clj*. Make it look like this (you'll add more later):

```
(ns the-divine-cheese-code.visualization.svg)

(defn latlng->point
  "Convert lat/lng map to comma-separated string"
  [latlng]
  (str (:lng latlng) "," (:lat latlng)))

(defn points
  [locations]
  (clojure.string/join " " (map latlng->point locations)))
```

This defines two functions, `latlng->point` and `points`, which you'll use to convert a seq of latitude/longitude coordinates into a string of points. To use this code from the *core.clj* file, you have to `require` it. `require` takes a symbol designating a namespace and ensures that the namespace exists and is ready to be used; in this case, when you call `(require 'the-divine-cheese-code.visualization.svg)`, Clojure reads and evaluates the corresponding file.

By evaluating the file, it creates the `the-divine-cheese-code.visualization.svg` namespace and defines the functions `latlng->point` and `points` within that namespace. Even though the file *svg.clj* is in your project's directory, Clojure doesn't automatically evaluate it when it runs your project; you have to explicitly tell Clojure that you want to use it.

After requiring the namespace, you can *refer* it so that you don't have to use fully qualified names to reference the functions. Go ahead and require `the-divine-cheese-code.visualization.svg` and add the `heists` seq to make *core.clj* match the listing:

```
(ns the-divine-cheese-code.core)
;; Ensure that the SVG code is evaluated
(require 'the-divine-cheese-code.visualization.svg)
;; Refer the namespace so that you don't have to use the
;; fully qualified name to reference svg functions
(refer 'the-divine-cheese-code.visualization.svg)

(def heists [{:location "Cologne, Germany"
              :cheese-name "Archbishop Hildebold's Cheese Pretzel"
              :lat 50.95
              :lng 6.97}
             {:location "Zurich, Switzerland"
              :cheese-name "The Standard Emmental"
              :lat 47.37
              :lng 8.55}
             {:location "Marseille, France"
              :cheese-name "Le Fromage de Cosquer"
              :lat 43.30
              :lng 5.37}
             {:location "Zurich, Switzerland"
              :cheese-name "The Lesser Emmental"
              :lat 47.37
              :lng 8.55}
             {:location "Vatican City"
              :cheese-name "The Cheese of Turin"
              :lat 41.90
              :lng 12.45}])

(defn -main
  [& args]
  (println (points heists)))
```

Now you have a seq of heist locations to work with and you can use functions from the `visualization.svg` namespace. The `main` function simply applies the `points` function to `heists`. If you run the project with `lein run`, you should see this:

```
50.95,6.97 47.37,8.55 43.3,5.37 47.37,8.55 41.9,12.45
```

Hooray! You're one step closer to catching that purloiner of the fermented curd! Using `require` successfully loaded `the-divine-cheese-code.visualization.svg` for use.

The details of `require` are actually a bit complicated, but for practical purposes you can think of `require` as telling Clojure the following:

1. Do nothing if you've already called `require` with this symbol (`the-divine-cheese-code.visualization.svg`).
2. Otherwise, find the file that corresponds to this symbol using the rules described in "The Relationship Between File Paths and Namespace Names" on page 133. In this case, Clojure finds `src/the_divine_cheese_code/visualization/svg.clj`.

Read and evaluate the contents of that file. Clojure expects the file to declare a namespace corresponding to its path (which ours does).

`require` also lets you alias a namespace when you require it, using `:as` or `alias`. This:

```
(require '[the-divine-cheese-code.visualization.svg :as svg])
```

is equivalent to this:

```
(require 'the-divine-cheese-code.visualization.svg)
(alias 'svg 'the-divine-cheese-code.visualization.svg)
```

You can now use the aliased namespace:

```
(svg/points heists)
; => "50.95,6.97 47.37,8.55 43.3,5.37 47.37,8.55 41.9,12.45"
```

Clojure provides another shortcut. Instead of calling `require` and `refer` separately, the function `use` does both. It's frowned upon to use `use` in production code, but it's handy when you're experimenting in the REPL and you want to quickly get your hands on some functions. For example, this:

```
(require 'the-divine-cheese-code.visualization.svg)
(refer 'the-divine-cheese-code.visualization.svg)
```

is equivalent to this:

```
(use 'the-divine-cheese-code.visualization.svg)
```

You can alias a namespace with `use` just like you can with `require`. This:

```
(require 'the-divine-cheese-code.visualization.svg)
(refer 'the-divine-cheese-code.visualization.svg)
(alias 'svg 'the-divine-cheese-code.visualization.svg)
```

is equivalent to the code in Listing 6-2, which also shows aliased namespaces being used in function calls.

```
(use '[the-divine-cheese-code.visualization.svg :as svg])
(= svg/points points)
; => true

(= svg/latlng->point latlng->point)
; => true
```

Listing 6-2: Sometimes it's handy to both use and alias a namespace.

It may seem redundant to alias a namespace with `use` here because `use` already refers the namespace (which lets you simply call `points` instead of `svg/points`). In certain situations, though, it's handy because `use` takes the same options as `refer` (`:only`, `:exclude`, `:as`, and `:rename`). You might want to alias a namespace with `use` when you've skipped referring a symbol. You could use this:

```
(require 'the-divine-cheese-code.visualization.svg)
(refer 'the-divine-cheese-code.visualization.svg :as :only ['points])
```

Or you could use the `use` form in Listing 6-3 (which also includes examples of how you can call functions).

```
(use '[the-divine-cheese-code.visualization.svg :as svg :only [points]])
(refer 'the-divine-cheese-code.visualization.svg :as :only ['points])
(= svg/points points)
; => true

;; We can use the alias to reach latlng->point
svg/latlng->point
; This doesn't throw an exception

;; But we can't use the bare name
latlng->point
; This does throw an exception!
```

;; why is there the quote syntax in REFER, but not in USE? is this a typo?

Listing 6-3: Aliasing a namespace after you use it lets you refer to symbols that you excluded.

If you try Listing 6-3 in a REPL and `latlng->point` doesn't throw an exception, it's because you referred `latlng->point` in Listing 6-2. You'll need to restart your REPL session for the code to behave as shown in Listing 6-3.

The takeaway here is that `require` and `use` load files and optionally `alias` or `refer` their namespaces. As you write Clojure programs and read code written by others, you might encounter even more ways of writing `require` and `use`, at which point it'll make sense to read Clojure's API docs (*http://clojure.org/libs/*) to understand what's going on. However, what you've learned so far about `require` and `use` should cover 95.3 percent of your needs.

The ns Macro

Now it's time to look at the `ns` macro. The tools covered so far—`in-ns`, `refer`, `alias`, `require`, and `use`—are most often used when you're playing in the REPL. In your source code files, you'll typically use the `ns` macro because it allows you to use the tools described so far succinctly and provides other useful functionality. In this section, you'll learn about how one `ns` call can incorporate `require`, `use`, `in-ns`, `alias`, and `refer`.

One useful task `ns` does is refer the `clojure.core` namespace by default. That's why you can call `println` from within `the-divine-cheese-code.core` without using the fully qualified name, `clojure.core/println`.

You can control what gets referred from `clojure-core` with `:refer-clojure`, which takes the same options as `refer`:

```
(ns the-divine-cheese-code.core
  (:refer-clojure :exclude [println]))
```

If you called this at the beginning of *divine_cheese_code.core.clj*, it would break your code, forcing you to use `clojure.core/println` within the `-main` function.

Within `ns`, the form `(:refer-clojure)` is called a *reference*. This might look weird to you. Is this reference a function call? A macro? What is it? You'll learn more about the underlying machinery in Chapter 7. For now, you just need to understand how each reference maps to function calls. For example, the preceding code is equivalent to this:

```
(in-ns 'the-divine-cheese-code.core)
(refer 'clojure.core :exclude ['println])
```

There are six possible kinds of references within `ns`:

- `(:refer-clojure)`
- `(:require)`
- `(:use)`
- `(:import)`
- `(:load)`
- `(:gen-class)`

`(:import)` and `(:gen-class)` are covered in Chapter 12. I won't cover `(:load)` because it is seldom used.

`(:require)` works a lot like the `require` function. For example, this:

```
(ns the-divine-cheese-code.core
  (:require the-divine-cheese-code.visualization.svg))
```

is equivalent to this:

```
(in-ns 'the-divine-cheese-code.core)
(require 'the-divine-cheese-code.visualization.svg)
```

Notice that in the `ns` form (unlike the `in-ns` function call), you don't have to quote your symbol with `'`. You never have to quote symbols within `ns`.

You can also `alias` a library that you `require` within `ns`, just like when you call the function. This:

```
(ns the-divine-cheese-code.core
  (:require [the-divine-cheese-code.visualization.svg :as svg]))
```

is equivalent to this:

```
(in-ns 'the-divine-cheese-code.core)
(require ['the-divine-cheese-code.visualization.svg :as 'svg])
```

You can require multiple libraries in a `(:require)` reference as follows. This:

```
(ns the-divine-cheese-code.core
  (:require [the-divine-cheese-code.visualization.svg :as svg]
            [clojure.java.browse :as browse]))
```

is equivalent to this:

```
(in-ns 'the-divine-cheese-code.core)
(require ['the-divine-cheese-code.visualization.svg :as 'svg])
(require ['clojure.java.browse :as 'browse])
```

However, one difference between the `(:require)` reference and the `require` function is that the reference also allows you to refer names. This:

```
(ns the-divine-cheese-code.core
  (:require [the-divine-cheese-code.visualization.svg :refer [points]]))
```

is equivalent to this:

```
(in-ns 'the-divine-cheese-code.core)
(require 'the-divine-cheese-code.visualization.svg)
(refer 'the-divine-cheese-code.visualization.svg :only ['points])
```

You can also refer all symbols (notice the `:all` keyword):

```
(ns the-divine-cheese-code.core
  (:require [the-divine-cheese-code.visualization.svg :refer :all]))
```

which is the same as doing this:

```
(in-ns 'the-divine-cheese-code.core)
(require 'the-divine-cheese-code.visualization.svg)
(refer 'the-divine-cheese-code.visualization.svg)
```

This is the preferred way to require code, alias namespaces, and refer symbols. It's recommended that you not use (:use), but since it's likely that you'll come across it, it's good to know how it works. You know the drill. This:

```
(ns the-divine-cheese-code.core
  (:use clojure.java.browse))
```

does this:

```
(in-ns 'the-divine-cheese-code.core)
(use 'clojure.java.browse)
```

whereas this:

```
(ns the-divine-cheese-code.core
  (:use [clojure.java browse io]))
```

does this:

```
(in-ns 'the-divine-cheese-code.core)
(use 'clojure.java.browse)
(use 'clojure.java.io)
```

Notice that when you follow :use with a vector, it takes the first symbol as the *base* and then calls use with each symbol that follows.

Oh my god, that's it! Now you can use ns like a pro! And you're going to need to, dammit, because that *voleur des fromages* (as they probably say in French) is still running amok! Remember him/her?!

To Catch a Burglar

We can't allow this plunderer of parmesan to make off with any more cheese! It's time to finish drawing lines based on the coordinates of the heists! That will surely reveal something!

Using the latitude coordinates for each heist, you'll connect the dots in an SVG image. But if you draw lines using the given coordinates, the result won't look right for two reasons. First, latitude coordinates ascend from south to north, whereas SVG y-coordinates ascend from top to bottom. In other words, you need to flip the coordinates or the drawing will be upside down.

Second, the drawing will be very small. To fix that, you'll zoom in on it by translating and scaling it. It's like turning a drawing that looks like Figure 6-1a into Figure 6-1b.

Figure 6-1: Flipping, translating, and scaling latitude coordinates to make an SVG image.

Honestly, this is all completely arbitrary and it's no longer directly related to code organization, but it's fun and I think you'll have a good time going through the code! Make your *svg.clj* file match Listing 6-4:

```
(ns the-divine-cheese-code.visualization.svg
  (:require [clojure.string :as s])
  (:refer-clojure :exclude [min max]))

❶ (defn comparator-over-maps
    [comparison-fn ks]
    (fn [maps]
❷     (zipmap ks
❸             (map (fn [k] (apply comparison-fn (map k maps)))
                  ks))))

❹ (def min (comparator-over-maps clojure.core/min [:lat :lng]))
  (def max (comparator-over-maps clojure.core/max [:lat :lng]))
```

Listing 6-4: Constructing map comparison functions

You define the `comparator-over-maps` function at ❶. This is probably the trickiest bit, so bear with me. `comparator-over-maps` is a function that returns a function. The returned function compares the values for the keys provided by the `ks` parameter using the supplied comparison function, `comparison-fn`.

You use `comparator-over-maps` to construct the `min` and `max` functions ❹, which you'll use to find the top-left and bottom-right corners of our drawing. Here's `min` in action:

```
(min [{:a 1 :b 3} {:a 5 :b 0}])
; => {:a 1 :b 0}
```

When you call `min`, it calls `zipmap` ❷, which takes two arguments, both seqs, and returns a new map. The elements of the first seq become the keys, and the elements of the second seq become the values:

```
(zipmap [:a :b] [1 2])
; => {:a 1 :b 2}
```

At ❷, the first argument to `zipmap` is `ks`, so the elements of `ks` will be the keys of the returned map. The second argument is the result of the map call at ❸. That map call actually performs the comparison.

Finally, at ❹ you use `comparator-over-maps` to create the comparison functions. If you think of the drawing as being inscribed in a rectangle, `min` is the corner of the rectangle closest to (0, 0) and `max` is the corner farthest from it.

Here's the next part of the code:

```
❺ (defn translate-to-00
    [locations]
    (let [mincoords (min locations)]
      (map #(merge-with - % mincoords) locations)))

❻ (defn scale
    [width height locations]
    (let [maxcoords (max locations)
          ratio {:lat (/ height (:lat maxcoords))
                 :lng (/ width (:lng maxcoords))}]
      (map #(merge-with * % ratio) locations)))
```

`translate-to-00`, defined at ❺, works by finding the `min` of our locations and subtracting that value from each location. It uses `merge-with`, which works like this:

```
(merge-with - {:lat 50 :lng 10} {:lat 5 :lng 5})
; => {:lat 45 :lng 5}
```

Then we define the function `scale` at ❻, which multiplies each point by the ratio between the maximum latitude and longitude and the desired height and width.

Here's the rest of the code for *svg.clj*:

```
(defn latlng->point
  "Convert lat/lng map to comma-separated string"
  [latlng]
  (str (:lng latlng) "," (:lat latlng)))

(defn points
  "Given a seq of lat/lng maps, return string of points joined by space"
  [locations]
  (s/join " " (map latlng->point locations)))

(defn line
  [points]
  (str "<polyline points=\"" points "\" />"))

(defn transform
  "Just chains other functions"
  [width height locations]
  (->> locations
       translate-to-00
       (scale width height)))
```

```
(defn xml
  "svg 'template', which also flips the coordinate system"
  [width height locations]
  (str "<svg height=\"" height "\" width=\"" width "\">"
       ;; These two <g> tags flip the coordinate system
       "<g transform=\"translate(0," height ")\">"
       "<g transform=\"scale(1,-1)\">"
       (-> (transform width height locations)
           points
           line)
       "</g></g>"
       "</svg>"))
```

The functions here are pretty straightforward. They just take `{:lat x :lng y}` maps and transform them so that an SVG can be created. `latlng->point` returns a string that can be used to define a point in SVG markup. `points` converts a seq of `lat/lng` maps into a space-separated string of points. `line` returns the SVG markup for a line that connects all given space-separated strings of points. `transform` takes a seq of locations, translates them so they start at the point (0, 0), and scales them to the given width and height. Finally, `xml` produces the markup for displaying the given locations using SVG.

With *svg.clj* all coded up, now make *core.clj* look like this:

```
(ns the-divine-cheese-code.core
  (:require [clojure.java.browse :as browse]
            [the-divine-cheese-code.visualization.svg :refer [xml]])
  (:gen-class))

(def heists [{:location "Cologne, Germany"
              :cheese-name "Archbishop Hildebold's Cheese Pretzel"
              :lat 50.95
              :lng 6.97}
             {:location "Zurich, Switzerland"
              :cheese-name "The Standard Emmental"
              :lat 47.37
              :lng 8.55}
             {:location "Marseille, France"
              :cheese-name "Le Fromage de Cosquer"
              :lat 43.30
              :lng 5.37}
             {:location "Zurich, Switzerland"
              :cheese-name "The Lesser Emmental"
              :lat 47.37
              :lng 8.55}
             {:location "Vatican City"
              :cheese-name "The Cheese of Turin"
              :lat 41.90
              :lng 12.45}])
```

```
(defn url
  [filename]
  (str "file:///"
       (System/getProperty "user.dir")
       "/"
       filename))

(defn template
  [contents]
  (str "<style>polyline { fill:none; stroke:#5881d8; stroke-width:3}</style>"
       contents))

(defn -main
  [& args]
  (let [filename "map.html"]
    (->> heists
         (xml 50 100)
         template
         (spit filename))
    (browse/browse-url (url filename))))
```

Nothing too complicated is going on here. Within `-main` you build up the drawing using the `xml` and `template` functions, write the drawing to a file with `spit`, and then open it with `browse/browse-url`. You should try that now! Run `lein run` and you'll see something that looks like Figure 6-2.

Figure 6-2: The final SVG of the heist pattern!

Wait a minute . . . that looks a lot like . . . that looks a lot like a lambda. Clojure's logo is a lambda . . . oh my god! Clojure, it was you this whole time!

Summary

You learned a lot in this chapter. At this point, you should have all the tools you need to start organizing your projects. You now know that namespaces organize maps between symbols and vars, and that vars are references to Clojure objects (data structures, functions, and so on). `def` stores an object and updates the current namespace with a map between a symbol and a var that points to the object. You can create private functions with `defn-`.

Clojure lets you create namespaces with `create-ns`, but often it's more useful to use `in-ns`, which switches to the namespace as well. You'll probably only use these functions in the REPL. When you're in the REPL, you're always *in* the current namespace. When you're defining namespaces in a file rather than the REPL, you should use the `ns` macro, and there's a one-to-one relationship between a namespace and its path on the filesystem.

You can refer to objects in other namespaces by using the fully qualified name, like `cheese.taxonomy/cheddars`. `refer` lets you use names from other namespaces without having to fully qualify them, and `alias` lets you use a shorter name for a namespace when you're writing out a fully qualified name.

`require` and `use` ensure that a namespace exists and is ready to be used, and optionally let you `refer` and `alias` the corresponding namespaces. You should use `ns` to call `require` and `use` in your source files. *https://gist.github.com/ghoseb/287710/* is a great reference for all the vagaries of using `ns`.

Lastly and most importantly, it ain't easy being cheesy.

7

CLOJURE ALCHEMY: READING, EVALUATION, AND MACROS

The philosopher's stone, along with the elixir of life and Viagra, is one of the most well-known specimens of alchemical lore, pursued for its ability to transmute lead into gold. Clojure, however, offers a tool that makes the philosopher's stone look like a mere trinket: the *macro*.

Macros allow you to transform arbitrary expressions into valid Clojure, so you can extend the language itself to fit your needs. And you don't even have to be a wizened old dude or lady in a robe to use them!

To get just a sip of this power, consider this trivial macro:

```
(defmacro backwards
  [form]
  (reverse form))

(backwards (" backwards" " am" "I" str))
; => "I am backwards"
```

The `backwards` macro allows Clojure to successfully evaluate the expression `(" backwards" " am" "I" str)`, even though it doesn't follow Clojure's built-in syntax rules, which require an expression's operand to appear first (not to mention the rule that an expression not be written in reverse order). Without `backwards`, the expression would fail harder than millennia of alchemists ironically spending their entire lives pursuing an impossible means of achieving immortality. With `backwards`, *you've created your own syntax!* You've extended Clojure so you can write code however you please! Better than turning lead into gold, I tell you!

This chapter gives you the conceptual foundation you need to go mad with power writing your own macros. It explains the elements of Clojure's evaluation model: the *reader*, the *evaluator*, and the *macro expander*. It's like the periodic table of Clojure elements. Think of how the periodic table reveals the properties of atoms: elements in the same column behave similarly because they have the same nuclear charge. Without the periodic table and its underlying theory, we'd be in the same position as the alchemists of yore, mixing stuff together randomly to see what blows up. But with a deeper understanding of the elements, you can see why stuff blows up and learn how to blow stuff up on purpose.

An Overview of Clojure's Evaluation Model

Clojure (like all Lisps) has an evaluation model that differs from most other languages: it has a two-phase system where it *reads* textual source code, producing Clojure data structures. These data structures are then *evaluated*: Clojure traverses the data structures and performs actions like function application or var lookup based on the type of the data structure. For example, when Clojure reads the text `(+ 1 2)`, the result is a list data structure whose first element is a `+` symbol, followed by the numbers 1 and 2. This data structure is passed to Clojure's evaluator, which looks up the function corresponding to `+` and applies that function to 1 and 2.

Languages that have this relationship between source code, data, and evaluation are called *homoiconic*. (Incidentally, if you say *homoiconic* in front of your bathroom mirror three times with the lights out, the ghost of John McCarthy appears and hands you a parenthesis.) Homoiconic languages empower you to reason about your code as a set of data structures that you can manipulate programmatically. To put this into context, let's take a jaunt through the land of compilation.

Programming languages require a compiler or interpreter for translating the code you write, which consists of Unicode characters, into something else: machine instructions, code in another programming language, whatever. During this process, the

compiler constructs an *abstract syntax tree (AST)*, which is a data structure that represents your program. You can think of the AST as the input to the *evaluator*, which you can think of as a function that traverses the tree to produce the machine code or whatever as its output.

So far this sounds a lot like what I described for Clojure. However, in most languages the AST's data structure is inaccessible within the programming language; the programming language space and the compiler space are forever separated, and never the twain shall meet. Figure 7-1 shows how you might visualize the compilation process for an expression in a non-Lisp programming language.

Figure 7-1: Evaluation in a non-Lisp programming language

But Clojure is different, because Clojure is a Lisp and Lisps are hotter than a stolen tamale. Instead of evaluating an AST that's represented as some inaccessible internal data structure, Lisps evaluate native data structures. Clojure still evaluates tree structures, but the trees are structured using Clojure lists and the nodes are Clojure values.

Lists are ideal for constructing tree structures. The first element of a list is treated as the root, and each subsequent element is treated as a branch. To create a nested tree, you can just use nested lists, as shown in Figure 7-2.

Figure 7-2: Lists can easily be treated as trees.

First, Clojure's *reader* converts the text `(+ 1 (* 6 7))` into a nested list. (You'll learn more about the reader in the next section.) Then, Clojure's evaluator takes that data as input and produces a result. (It also compiles Java Virtual Machine (JVM) bytecode, which you'll learn about in Chapter 12. For now, we'll just focus on the evaluation model on a conceptual level.)

With this in mind, Figure 7-3 shows what Clojure's evaluation process looks like.

S-EXPRESSIONS

In your Lisp adventures, you'll come across resources that explain that Lisps evaluate *s-expressions*. I avoid that term here because it's ambiguous: you'll see it used to refer to both the actual data object that gets evaluated and the source code that represents that data. Using the same term for two different components of Lisp evaluation (code and data) obscures what's important: your text represents native data structures, and Lisps evaluate native data structures, which is unique and awesome. For a great treatment of s-expressions, check out *http://www.gigamonkeys.com/book/syntax-and-semantics.html*.

Figure 7-3: Evaluation in Clojure

However, the evaluator doesn't actually care where its input comes from; it doesn't have to come from the reader. As a result, you can send your program's data structures directly to the Clojure evaluator with `eval`. Behold!

```
(def addition-list (list + 1 2))
(eval addition-list)
; => 3
```

That's right, baby! Your program just evaluated a Clojure list. You'll read all about Clojure's evaluation rules soon, but briefly, this is what happened: when Clojure evaluated the list, it looked up the list that `addition-list` refers to; then it looked up the function corresponding to the `+` symbol; and then it called that function with `1` and `2` as arguments, returning `3`. The data structures of your running program and those

of the evaluator live in the same space, and the upshot is that you can use the full power of Clojure and all the code you've written to construct data structures for evaluation:

```
(eval (concat addition-list [10]))
; => 13

(eval (list 'def 'lucky-number (concat addition-list [10])))
; => #'user/lucky-number

lucky-number
; => 13
```

Figure 7-4 shows the lists you sent to the evaluator in these two examples.

Figure 7-4: The lists you evaluated

Your program can talk directly to its own evaluator, using its own functions and data to modify itself as it runs! Are you going mad with power yet? I hope so! Hold on to some of your sanity, though, because there's still more to learn.

So Clojure is homoiconic: it represents abstract syntax trees using lists, and you write textual representations of lists when you write Clojure code. Because the code you write represents data structures that you're used to manipulating and the evaluator consumes those data structures, it's easy to reason about how to programmatically modify your program.

Macros are what allow you to perform those manipulations easily. The rest of this chapter covers Clojure's reader and evaluation rules in detail to give you a precise understanding of how macros work.

The Reader

The reader converts the textual source code you save in a file or enter in the REPL into Clojure data structures. It's like a translator between the human world of Unicode characters and Clojure's world of lists, vectors, maps, symbols, and other data structures. In this section, you'll interact directly with the reader and learn how a handy feature, the *reader macro*, lets you write code more succinctly.

Reading

To understand reading, let's first take a close look at how Clojure handles the text you type in the REPL. First, the REPL prompts you for text:

```
user=>
```

Then you enter a bit of text. Maybe something like this:

```
user=> (str "To understand what recursion is," " you must first understand recursion.")
```

That text is really just a sequence of Unicode characters, but it's meant to represent a combination of Clojure data structures. This textual representation of data structures is called a *reader form*. In this example, the form represents a list data structure that contains three more forms: the `str` symbol and two strings.

Once you type those characters into the prompt and press ENTER, that text goes to the reader (remember REPL stands for read-eval-print-loop). Clojure reads the stream of characters and internally produces the corresponding data structures. It then evaluates the data structures and prints the textual representation of the result:

```
"To understand what recursion is, you must first understand recursion."
```

Reading and evaluation are discrete processes that you can perform independently. One way to interact with the reader directly is by using the `read-string` function. `read-string` takes a string as an argument and processes it using Clojure's reader, returning a data structure:

```
(read-string "(+ 1 2)")
; => (+ 1 2)

(list? (read-string "(+ 1 2)"))
; => true

(conj (read-string "(+ 1 2)") :zagglewag)
; => (:zagglewag + 1 2)
```

In the first example, `read-string` reads the string representation of a list containing a plus symbol and the numbers 1 and 2. The return value is an actual list, as proven by the second example. The last example uses `conj` to

prepend a keyword to the list. The takeaway is that reading and evaluating are independent of each other. You can read text without evaluating it, and you can pass the result to other functions. You can also evaluate the result, if you want:

```
(eval (read-string "(+ 1 2)"))
; => 3
```

In all the examples so far, there's been a one-to-one relationship between the reader form and the corresponding data structures. Here are more examples of simple reader forms that directly map to the data structures they represent:

`()` A list reader form

`str` A symbol reader form

`[1 2]` A vector reader form containing two number reader forms

`{:sound "hoot"}` A map reader form with a keyword reader form and string reader form

However, the reader can employ more complex behavior when converting text to data structures. For example, remember anonymous functions?

```
(#(+ 1 %) 3)
; => 4
```

Well, try this out:

```
(read-string "#(+ 1 %)")
; => (fn* [p1__423#] (+ 1 p1__423#))
```

Whoa! This is not the one-to-one mapping that we're used to. Reading `#(+ 1 %)` somehow resulted in a list consisting of the `fn*` symbol, a vector containing a symbol, and a list containing three elements. What just happened?

Reader Macros

I'll answer my own question: the reader used a *reader macro* to transform `#(+ 1 %)`. Reader macros are sets of rules for transforming text into data structures. They often allow you to represent data structures in more compact ways because they take an abbreviated reader form and expand it into a full form. They're designated by *macro characters*, like `'` (the single quote), `#`, and `@`. They're also completely different from the macros we'll get to later. So as not to get the two confused, I'll always refer to reader macros using the full term *reader macros*.

For example, you can see how the quote reader macro expands the single quote character here:

```
(read-string "'(a b c)")
; => (quote (a b c))
```

When the reader encounters the single quote, it expands it to a list whose first member is the symbol `quote` and whose second member is the data structure that followed the single quote. The `deref` reader macro works similarly for the `@` character:

```
(read-string "@var")
; => (clojure.core/deref var)
```

Reader macros can also do crazy stuff like cause text to be ignored. The semicolon designates the single-line comment reader macro:

```
(read-string "; ignore!\n(+ 1 2)")
; => (+ 1 2)
```

And that's the reader! Your humble companion, toiling away at transforming text into data structures. Now let's look at how Clojure evaluates those data structures.

The Evaluator

You can think of Clojure's evaluator as a function that takes a data structure as an argument, processes the data structure using rules corresponding to the data structure's type, and returns a result. To evaluate a symbol, Clojure looks up what the symbol refers to. To evaluate a list, Clojure looks at the first element of the list and calls a function, macro, or special form. Any other values (including strings, numbers, and keywords) simply evaluate to themselves.

For example, let's say you've typed `(+ 1 2)` in the REPL. Figure 7-5 shows a diagram of the data structure that gets sent to the evaluator.

Figure 7-5: The data structure for `(+ 1 2)`

Because it's a list, the evaluator starts by evaluating the first element in the list. The first element is the plus symbol, and the evaluator resolves that by returning the corresponding function. Because the first element in the list is a function, the evaluator evaluates each of the operands. The operands 1 and 2 evaluate to themselves because they're not lists or symbols. Then the evaluator calls the addition function with 1 and 2 as the operands, and returns the result.

The rest of this section explains the evaluator's rules for each kind of data structure more fully. To show how the evaluator works, we'll just run each example in the REPL. Keep in mind that the REPL first reads your text to get a data structure, then sends that data structure to the evaluator, and then prints the result as text.

DATA

I write about how Clojure evaluates data structures in this chapter, but that's imprecise. Technically, *data structure* refers to some kind of collection, like a linked list or b-tree, or whatever, but I also use the term to refer to scalar (singular, noncollection) values like symbols and numbers. I considered using the term *data objects* but didn't want to imply object-oriented programming, or using just *data* but didn't want to confuse that with data as a concept. So, *data structure* it is, and if you find this offensive, I will give you a thousand apologies, thoughtfully organized in a Van Emde Boas tree.

These Things Evaluate to Themselves

Whenever Clojure evaluates data structures that aren't a list or symbol, the result is the data structure itself:

```
true
; => true

false
; => false

{}
; => {}

:huzzah
; => :huzzah
```

Empty lists evaluate to themselves, too:

```
()
; => ()
```

Symbols

One of your fundamental tasks as a programmer is creating abstractions by associating names with values. You learned how to do this in Chapter 3 by using `def`, `let`, and function definitions. Clojure uses *symbols* to name functions, macros, data, and anything else you can use, and evaluates them by *resolving* them. To resolve a symbol, Clojure traverses any bindings you've created and then looks up the symbol's entry in a namespace mapping, which you learned about in Chapter 6. Ultimately, a symbol resolves to either a *value* or a *special form*—a built-in Clojure operator that provides fundamental behavior.

In general, Clojure resolves a symbol by:

1. Looking up whether the symbol names a special form. If it doesn't . . .
2. Looking up whether the symbol corresponds to a local binding. If it doesn't . . .
3. Trying to find a namespace mapping introduced by `def`. If it doesn't . . .
4. Throwing an exception

Let's first look at a symbol resolving to a special form. Special forms, like `if`, are always used in the context of an operation; they're always the first element in a list:

```
(if true :a :b)
; => :a
```

In this case, `if` is a special form and it's being used as an operator. If you try to refer to a special form outside of this context, you'll get an exception:

```
if
; => CompilerException java.lang.RuntimeException: Unable to resolve symbol:
if in this context, compiling:(NO_SOURCE_PATH:0:0)
```

Next, let's evaluate some local bindings. A *local binding* is any association between a symbol and a value that wasn't created by `def`. In the next example, the symbol `x` is bound to 5 using `let`. When the evaluator resolves `x`, it resolves the *symbol* `x` to the *value* 5:

```
(let [x 5]
  (+ x 3))
; => 8
```

Now if we create a namespace mapping of `x` to 15, Clojure resolves it accordingly:

```
(def x 15)
(+ x 3)
; => 18
```

In the next example, `x` is mapped to 15, but we introduce a local binding of `x` to 5 using `let`. So `x` is resolved to 5:

```
(def x 15)
(let [x 5]
  (+ x 3))
; => 8
```

You can nest bindings, in which case the most recently defined binding takes precedence:

```
(let [x 5]
  (let [x 6]
    (+ x 3)))
; => 9
```

Functions also create local bindings, binding parameters to arguments within the function body. In this next example, `exclaim` is mapped to a function. Within the function body, the parameter name `exclamation` is bound to the argument passed to the function:

```
(defn exclaim
  [exclamation]
  (str exclamation "!"))

(exclaim "Hadoken")
; => "Hadoken!"
```

Finally, in this last example, `map` and `inc` both refer to functions:

```
(map inc [1 2 3])
; => (2 3 4)
```

When Clojure evaluates this code, it first evaluates the `map` symbol, looking up the corresponding function and applying it to its arguments. The symbol `map` refers to the map function, but it shouldn't be confused with the function itself. The `map` symbol is still a data structure, the same way that the string `"fried salad"` is a data structure, but it's not the same as the function itself:

```
(read-string ("+"))
; => +

(type (read-string "+"))
; => clojure.lang.Symbol

(list (read-string "+") 1 2)
; => (+ 1 2)
```

In these examples, you're interacting with the plus symbol, `+`, as a data structure. You're not interacting with the addition function that it refers to. If you evaluate it, Clojure looks up the function and applies it:

```
(eval (list (read-string "+") 1 2))
; => 3
```

On their own, symbols and their referents don't actually do anything; Clojure performs work by evaluating lists.

Lists

If the data structure is an empty list, it evaluates to an empty list:

```
(eval (read-string "()"))
; => ()
```

Otherwise, it is evaluated as a *call* to the first element in the list. The way the call is performed depends on the nature of that first element.

Function Calls

When performing a function call, each operand is fully evaluated and then passed to the function as an argument. In this example, the `+` symbol resolves to a function:

```
(+ 1 2)
; => 3
```

Clojure sees that the list's head is a function, so it proceeds to evaluate the rest of the elements in the list. The operands 1 and 2 both evaluate to themselves, and after they're evaluated, Clojure applies the addition function to them.

You can also nest function calls:

```
(+ 1 (+ 2 3))
; => 6
```

Even though the second argument is a list, Clojure follows the same process here: look up the `+` symbol and evaluate each argument. To evaluate the list `(+ 2 3)`, Clojure resolves the first member to the addition function and proceeds to evaluate each of the arguments. In this way, evaluation is recursive.

Special Forms

You can also call *special forms*. In general, special forms are special because they implement core behavior that can't be implemented with functions. For example:

```
(if true 1 2)
; => 1
```

Here, we ask Clojure to evaluate a list beginning with the symbol `if`. That `if` symbol gets resolved to the `if` special form, and Clojure calls that special form with the operands `true`, `1`, and `2`.

Special forms don't follow the same evaluation rules as normal functions. For example, when you call a function, each operand gets evaluated.

However, with `if` you don't want each operand to be evaluated. You only want certain operands to be evaluated, depending on whether the condition is true or false.

Another important special form is `quote`. You've seen lists represented like this:

```
'(a b c)
```

As you saw in "The Reader" on page 153, this invokes a reader macro so that we end up with this:

```
(quote (a b c))
```

Normally, Clojure would try to resolve the `a` symbol and then call it because it's the first element in a list. The `quote` special form tells the evaluator, "Instead of evaluating my next data structure like normal, just return the data structure itself." In this case, you end up with a list consisting of the symbols `a`, `b`, and `c`.

`def`, `let`, `loop`, `fn`, `do`, and `recur` are all special forms as well. You can see why: they don't get evaluated the same way as functions. For example, normally when the evaluator evaluates a symbol, it resolves that symbol, but `def` and `let` obviously don't behave that way. Instead of resolving symbols, they actually create associations between symbols and values. So the evaluator receives a combination of data structures from the reader, and it goes about resolving the symbols and calling the functions or special forms at the beginning of each list. But there's more! You can also place a *macro* at the beginning of a list instead of a function or a special form, and this can give you tremendous power over how the rest of the data structures are evaluated.

Macros

Hmm . . . Clojure evaluates data structures—the same data structures that we write and manipulate in our Clojure programs. Wouldn't it be awesome if we could use Clojure to manipulate the data structures that Clojure evaluates? Yes, yes it would! And guess what? You can do this with macros! Did your head just explode? Mine did!

To get an idea of what macros do, let's look at some code. Say we want to write a function that makes Clojure read infix notation (such as `1 + 1`) instead of its normal notation with the operator first (`+ 1 1`). This example is *not* a macro. Rather, it merely shows that you can write code using infix notation and then use Clojure to transform it so it will actually execute. First, create a list that represents infix addition:

```
(read-string "(1 + 1)")
; => (1 + 1)
```

Clojure will throw an exception if you try to make it evaluate this list:

```
(eval (read-string "(1 + 1)"))
; => ClassCastException java.lang.Long cannot be cast to clojure.lang.IFn
```

However, `read-string` returns a list, and you can use Clojure to reorganize that list into something it *can* successfully evaluate:

```
(let [infix (read-string "(1 + 1)")]
  (list (second infix) (first infix) (last infix)))
; => (+ 1 1)
```

If you `eval` this, it returns `2`, just as you'd expect:

```
(eval
 (let [infix (read-string "(1 + 1)")]
   (list (second infix) (first infix) (last infix))))
; => 2
```

This is cool, but it's also quite clunky. That's where macros come in. Macros give you a convenient way to manipulate lists before Clojure evaluates them. Macros are a lot like functions: they take arguments and return a value, just like a function would. They work on Clojure data structures, just like functions do. What makes them unique and powerful is the way they fit in to the evaluation process. They are executed in between the reader and the evaluator—so they can manipulate the data structures that the reader spits out and transform with those data structures before passing them to the evaluator.

Let's look at an example:

```
(defmacro ignore-last-operand
  [function-call]
  (butlast function-call))

❶ (ignore-last-operand (+ 1 2 10))
; => 3

;; This will not print anything
(ignore-last-operand (+ 1 2 (println "look at me!!!")))
; => 3
```

At ❶ the macro `ignore-last-operand` receives the list `(+ 1 2 10)` as its argument, *not* the value `13`. This is very different from a function call, because function calls always evaluate all of the arguments passed in, so there is no possible way for a function to reach into one of its operands and alter or ignore it. By contrast, when you call a macro, the operands are *not* evaluated. In particular, symbols are not resolved; they are passed as symbols. Lists are not evaluated either; that is, the first element in the list is not called as a function, special form, or macro. Rather, the unevaluated list data structure is passed in.

Another difference is that the data structure returned by a function is *not* evaluated, but the data structure returned by a macro *is*. The process of determining the return value of a macro is called *macro expansion*, and you can use the function `macroexpand` to see what data structure a macro returns before that data structure is evaluated. Note that you have to quote the form that you pass to `macroexpand`:

```
(macroexpand '(ignore-last-operand (+ 1 2 10)))
; => (+ 1 2)

(macroexpand '(ignore-last-operand (+ 1 2 (println "look at me!!!"))))
; => (+ 1 2)
```

As you can see, both expansions result in the list `(+ 1 2)`. When this list is evaluated, as in the previous example, the result is `3`.

Just for fun, here's a macro for doing simple infix notation:

```
(defmacro infix
  [infixed]
  (list (second infixed)
        (first infixed)
        (last infixed)))

(infix (1 + 2))
; => 3
```

The best way to think about this whole process is to picture a phase between reading and evaluation: the *macro expansion* phase. Figure 7-6 shows how you can visualize the entire evaluation process for `(infix (1 + 2))`.

And that's how macros fit into the evaluation process. But why would you want to do this? The reason is that macros allow you to transform an arbitrary data structure like `(1 + 2)` into one that can Clojure can evaluate, `(+ 1 2)`. That means *you can use Clojure to extend itself* so you can write programs however you please. In other words, macros enable *syntactic abstraction*. Syntactic abstraction may sound a bit abstract (ha ha!), so let's explore that a little.

Figure 7-6: The full evaluation process for `(infix (1 + 2))`

Syntactic Abstraction and the -> Macro

Often, Clojure code consists of a bunch of nested function calls. For example, I use the following function in one of my projects:

```
(defn read-resource
  "Read a resource into a string"
  [path]
  (read-string (slurp (clojure.java.io/resource path))))
```

To understand the function body, you have to find the innermost form, in this case (`clojure.java.io/resource path`), and then work your way outward from right to left to see how the result of each function gets passed to another function. This right-to-left flow is opposite of what non-Lisp programmers are used to. As you get used to writing in Clojure, this kind of code gets easier and easier to understand. But if you want to translate Clojure code so you can read it in a more familiar, left-to-right, top-to-bottom manner, you can use the built-in `->` macro, which is also known as the *threading* or *stabby* macro. It lets you rewrite the preceding function like this:

```
(defn read-resource
  [path]
  (-> path
      clojure.java.io/resource
      slurp
      read-string))
```

You can read this as a pipeline that goes from top to bottom instead of from inner parentheses to outer parentheses. First, `path` gets passed to `io/resource`, then the result gets passed to `slurp`, and finally the result of that gets passed to `read-string`.

These two ways of defining `read-resource` are entirely equivalent. However, the second one might be easier understand because we can approach it from top to bottom, a direction we're used to. The `->` also lets us omit parentheses, which means there's less visual noise to contend with. This is a *syntactic abstraction* because it lets you write code in a syntax that's different from Clojure's built-in syntax but is preferable for human consumption. Better than lead into gold!!!

Summary

In this chapter, you learned about Clojure's evaluation process. First, the reader transforms text into Clojure data structures. Next, the macro expander transforms those data structures with macros, converting your custom syntax into syntactically valid data structures. Finally, those data structures get sent to the evaluator. The evaluator processes data structures based on their type: symbols are resolved to their referents; lists result in function, macro, or special form calls; and everything else evaluates to itself.

The coolest thing about this process is that it allows you to use Clojure to expand its own syntax. This process is made easier because Clojure is homoiconic: its text represents data structures, and those data structures represent abstract syntax trees, allowing you to more easily reason about how to construct syntax-expanding macros.

With all these new concepts in your brainacles, you're now ready to blow stuff up on purpose, just like I promised. The next chapter will teach you everything you need to know about writing macros. Hold on to your socks or they're liable to get knocked off!

Exercises

These exercises focus on reading and evaluation. Chapter 8 has exercises for writing macros.

1. Use the `list` function, quoting, and `read-string` to create a list that, when evaluated, prints your first name and your favorite sci-fi movie.
2. Create an infix function that takes a list like `(1 + 3 * 4 - 5)` and transforms it into the lists that Clojure needs in order to correctly evaluate the expression using operator precedence rules.

8

WRITING MACROS

When I was 18, I got a job as a night auditor at a hotel in Santa Fe, New Mexico, working four nights a week from 11 PM till 7 AM. After a few months of this sleepless schedule, my emotions took on a life of their own. One night, at about 3 AM, I was watching an infomercial for a product claiming to restore men's hair. As I watched the story of a formerly bald individual, I became overwhelmed with sincere joy. "At last!" my brain gushed. "This man has gotten the love and success he deserves! What an incredible product, giving hope to the hopeless!"

Since then I've found myself wondering if I could somehow re-create the emotional abandon and appreciation for life induced by chronic sleep deprivation. Some kind of potion, perhaps—a couple quaffs to unleash my inner Richard Simmons, but not for too long.

Just as a potion would allow me to temporarily alter my fundamental nature, macros allow you to modify Clojure in ways that just aren't possible with other languages. With macros, you can extend Clojure to suit your problem space, building up the language.

In this chapter, you'll thoroughly examine how to write macros, starting with basic examples and moving up in complexity. You'll close by donning your make-believe cap and using macros to validate customer orders in your imaginary online potion store.

By the end of the chapter, you'll understand all the tools you'll use to write macros: quote, syntax quote, unquote, unquote splicing (aka the piñata tool), and gensym. You'll also learn about the dangers lying in wait for unsuspecting macro authors: double evaluation, variable capture, and macro infection.

Macros Are Essential

Before you start writing macros, I want to help you put them in the proper context. Yes, macros are cooler than a polar bear's toenails, but you shouldn't think of macros as some esoteric tool you pull out when you feel like getting extra fancy with your code. In fact, macros allow Clojure to derive a lot of its built-in functionality from a tiny core of functions and special forms. Take `when`, for example. `when` has this general form:

```
(when boolean-expression
  expression-1
  expression-2
  expression-3
  ...
  expression-x)
```

You might think that `when` is a special form like `if`. Well guess what? It's not! In most other languages, you can only create conditional expressions using special keywords, and there's no way to create your own conditional operators. However, `when` is actually a macro.

In this macro expansion, you can see that `when` is implemented in terms of `if` and `do`:

```
(macroexpand '(when boolean-expression
                expression-1
                expression-2
                expression-3))
; => (if boolean-expression
       (do expression-1
           expression-2
           expression-3))
```

This shows that macros are an integral part of Clojure development—they're even used to provide fundamental operations. Macros aren't reserved for exotic special cases; you should think of macro writing as just another tool in your tool satchel. As you learn to write your own macros, you'll see how they allow you to extend the language even further so that it fits the shape of your particular problem domain.

Anatomy of a Macro

Macro definitions look much like function definitions. They have a name, an optional document string, an argument list, and a body. The body will almost always return a list. This makes sense because macros are a way of transforming a data structure into a form Clojure can evaluate, and Clojure uses lists to represent function calls, special form calls, and macro calls. You can use any function, macro, or special form within the macro body, and you call macros just like you would a function or special form.

As an example, here's our old friend the `infix` macro:

```
(defmacro infix
  "Use this macro when you pine for the notation of your childhood"
  [infixed]
  (list (second infixed) (first infixed) (last infixed)))
```

This macro rearranges a list into the correct order for infix notation. Here's an example:

```
(infix (1 + 1))
; => 2
```

One key difference between functions and macros is that function arguments are fully evaluated before they're passed to the function, whereas macros receive arguments as unevaluated data. You can see this in the example. If you tried evaluating `(1 + 1)` on its own, you would get an exception. However, because you're making a macro call, the unevaluated list `(1 + 1)` is passed to `infix`. Then the macro can use `first`, `second`, and `last` to rearrange the list so Clojure can evaluate it:

```
(macroexpand '(infix (1 + 1)))
; => (+ 1 1)
```

By expanding the macro, you can see that `infix` rearranges `(1 + 1)` into `(+ 1 1)`. Handy!

You can also use argument destructuring in macro definitions, just like you can with functions:

```
(defmacro infix-2
  [[operand1 op operand2]]
  (list op operand1 operand2))
```

Destructuring arguments lets you succinctly bind values to symbols based on their position in a sequential argument. Here, `infix-2` takes a sequential data structure as an argument and destructures by position so the first value is named `operand1`, the second value is named `op`, and the third value is named `operand2` within the macro.

You can also create multiple-arity macros, and in fact the fundamental Boolean operations `and` and `or` are defined as macros. Here's `and`'s source code:

```
(defmacro and
  "Evaluates exprs one at a time, from left to right. If a form
  returns logical false (nil or false), and returns that value and
  doesn't evaluate any of the other expressions, otherwise it returns
  the value of the last expr. (and) returns true."
  {:added "1.0"}
  ([] true)
  ([x] x)
  ([x & next]
   `(let [and# ~x]
      (if and# (and ~@next) and#))))
```

There's a lot of stuff going on in this example, including the symbols `` ` `` and `~@`, which you'll learn about soon. What's important to realize for now is that there are three macro bodies here: a 0-arity macro body that always returns `true`, a 1-arity macro body that returns the operand, and an *n*-arity macro body that recursively calls itself. That's right: macros can be recursive, and they also can use rest args (`& next` in the *n*-arity macro body), just like functions.

Now that you're comfortable with the anatomy of macros, it's time to strap yourself to your thinking mast Odysseus-style and learn to write macro bodies.

Building Lists for Evaluation

Macro writing is all about building a list for Clojure to evaluate, and it requires a kind of inversion to your normal way of thinking. For one, you'll often need to quote expressions to get unevaluated data structures in your final list (we'll get back to that in a moment). More generally, you'll need to be extra careful about the difference between a *symbol* and its *value*.

Distinguishing Symbols and Values

Say you want to create a macro that takes an expression and both prints and returns its value. (This differs from `println` in that `println` always returns `nil`.) You want your macro to return lists that look like this:

```
(let [result expression]
  (println result)
  result)
```

Your first version of the macro might look like this, using the `list` function to create the list that Clojure should evaluate:

```
(defmacro my-print-whoopsie
  [expression]
  (list let [result expression]
        (list println result)
        result))
```

However, if you tried this, you'd get the exception `Can't take the value of a macro: #'clojure.core/let`. What's going on here?

The reason this happens is that your macro body tries to get the *value* that the *symbol* `let` refers to, whereas what you actually want to do is return the `let` symbol itself. There are other problems, too: you're trying to get the value of `result`, which is unbound, and you're trying to get the value of `println` instead of returning its symbol. Here's how you would write the macro to do what you want:

```
(defmacro my-print
  [expression]
  (list 'let ['result expression]
        (list 'println 'result)
        'result))
```

Here, you're quoting each symbol you want to use as a symbol by prefixing it with the single quote character, `'`. This tells Clojure to *turn off* evaluation for whatever follows, in this case preventing Clojure from trying to resolve the symbols and instead just returning the symbols. The ability to use quoting to turn off evaluation is central to writing macros, so let's give the topic its own section.

Simple Quoting

You'll almost always use quoting within your macros to obtain an unevaluated symbol. Let's go through a brief refresher on quoting and then see how you might use it in a macro.

First, here's a simple function call with no quoting:

```
(+ 1 2)
; => 3
```

If we add `quote` at the beginning, it returns an unevaluated data structure:

```
(quote (+ 1 2))
; => (+ 1 2)
```

Here in the returned list, + is a symbol. If we evaluate this plus symbol, it yields the plus function:

```
+
; => #<core$_PLUS_ clojure.core$_PLUS_@47b36583>
```

Whereas if we quote the plus symbol, it just yields the plus symbol:

```
(quote +)
; => +
```

Evaluating an unbound symbol raises an exception:

```
sweating-to-the-oldies
; => Unable to resolve symbol: sweating-to-the-oldies in this context
```

But quoting the symbol returns a symbol regardless of whether the symbol has a value associated with it:

```
(quote sweating-to-the-oldies)
; => sweating-to-the-oldies
```

The single quote character is a reader macro for `(quote x)`:

```
'(+ 1 2)
; => (+ 1 2)

'dr-jekyll-and-richard-simmons
; => dr-jekyll-and-richard-simmons
```

You can see quoting at work in the `when` macro. This is `when`'s actual source code:

```
(defmacro when
  "Evaluates test. If logical true, evaluates body in an implicit do."
  {:added "1.0"}
  [test & body]
  (list 'if test (cons 'do body)))
```

Notice that the macro definition quotes both `if` and `do`. That's because you want these symbols to be in the final list that `when` returns for evaluation. Here's an example of what that returned list might look like:

```
(macroexpand '(when (the-cows-come :home)
                (call me :pappy)
                (slap me :silly)))
; => (if (the-cows-come :home)
       (do (call me :pappy)
           (slap me :silly)))
```

Here's another example of source code for a built-in macro, this time for `unless`:

```
(defmacro unless
  "Inverted 'if'"
  [test & branches]
  (conj (reverse branches) test 'if))
```

Again, you have to quote `if` because you want the unevaluated symbol to be placed in the resulting list, like this one:

```
(macroexpand '(unless (done-been slapped? me)
                      (slap me :silly)
                      (say "I reckon that'll learn me")))
; => (if (done-been slapped? me)
       (say "I reckon that'll learn me")
       (slap me :silly))
```

In many cases, you'll use simple quoting like this when writing macros, but most often you'll use the more powerful syntax quote.

Syntax Quoting

So far, you've seen macros that build up lists by using the `list` function to create a list along with `'` (quote), and functions that operate on lists like `first`, `second`, `last`, and so on. Indeed, you could write macros that way until the cows come home. Sometimes, though, it leads to tedious and verbose code.

Syntax quoting returns unevaluated data structures, similar to normal quoting. However, there are two important differences. One difference is that syntax quoting will return the *fully qualified* symbols (that is, with the symbol's namespace included). Let's compare quoting and syntax quoting.

Quoting does not include a namespace if your code doesn't include a namespace:

```
'+
; => +
```

Write out the namespace, and it'll be returned by normal quote:

```
'clojure.core/+
; => clojure.core/+
```

Syntax quoting will always include the symbol's full namespace:

```
`+
; => clojure.core/+
```

Quoting a list recursively quotes all the elements:

```
'(+ 1 2)
; => (+ 1 2)
```

Syntax quoting a list recursively syntax quotes all the elements:

```
`(+ 1 2)
; => (clojure.core/+ 1 2)
```

The reason syntax quotes include the namespace is to help you avoid name collisions, a topic covered in Chapter 6.

The other difference between quoting and syntax quoting is that the latter allows you to *unquote* forms using the tilde, ~. It's kind of like kryptonite in that way: whenever Superman is around kryptonite, his powers disappear. Whenever a tilde appears within a syntax-quoted form, the syntax quote's power to return unevaluated, fully namespaced forms disappears. Here's an example:

```
`(+ 1 ~(inc 1))
; => (clojure.core/+ 1 2)
```

Because it comes after the tilde, `(inc 1)` is evaluated instead of being quoted. Without the unquote, syntax quoting returns the unevaluated form with fully qualified symbols:

```
`(+ 1 (inc 1))
; => (clojure.core/+ 1 (clojure.core/inc 1))
```

If you're familiar with string interpolation, you can think of syntax quoting/unquoting similarly. In both cases, you're creating a kind of template, placing a few variables within a larger, static structure. For example, in Ruby you can create the string `"Churn your butter, Jebediah"` through concatenation:

```
name = "Jebediah"
"Churn your butter, " + name + "!"
```

or through interpolation:

```
"Churn your butter, #{name}!"
```

In the same way that string interpolation leads to clearer and more concise code, syntax quoting and unquoting allow you to create lists more clearly and concisely. Compare using the `list` function, shown first, with using syntax quoting:

```
(list '+ 1 (inc 1))
; => (+ 1 2)
```

```
`(+ 1 ~(inc 1))
; => (clojure.core/+ 1 2)
```

As you can see, the syntax-quote version is more concise. Also, its visual form is closer to the final form of the list, making it easier to understand.

Using Syntax Quoting in a Macro

Now that you have a good handle on how syntax quoting works, take a look at the `code-critic` macro. You're going to write a more concise version using syntax quoting.

```
(defmacro code-critic
  "Phrases are courtesy Hermes Conrad from Futurama"
  [bad good]
  (list 'do
        (list 'println
              "Great squid of Madrid, this is bad code:"
              (list 'quote bad))
        (list 'println
              "Sweet gorilla of Manila, this is good code:"
              (list 'quote good))))

(code-critic (1 + 1) (+ 1 1))
; => Great squid of Madrid, this is bad code: (1 + 1)
; => Sweet gorilla of Manila, this is good code: (+ 1 1)
```

Just looking at all those tedious repetitions of `list` and single quotes makes me cringe. But if you rewrite `code-critic` using syntax quoting, you can make it sleek and concise:

```
(defmacro code-critic
  "Phrases are courtesy Hermes Conrad from Futurama"
  [bad good]
  `(do (println "Great squid of Madrid, this is bad code:"
                (quote ~bad))
       (println "Sweet gorilla of Manila, this is good code:"
                (quote ~good))))
```

In this case, you want to quote everything except for the symbols `good` and `bad`. In the original version, you have to quote each piece individually and explicitly place it in a list in an unwieldy fashion, just to prevent those two symbols from being quoted. With syntax quoting, you can just wrap the entire `do` expression in a quote and simply unquote the two symbols that you want to evaluate.

And thus concludes the introduction to the mechanics of writing a macro! Sweet sacred boa of Western and Eastern Samoa, that was a lot!

To sum up, macros receive unevaluated, arbitrary data structures as arguments and return data structures that Clojure evaluates. When

defining your macro, you can use argument destructuring just like you can with functions and `let` bindings. You can also write multiple-arity and recursive macros.

Most of the time, your macros will return lists. You can build up the list to be returned by using `list` functions or by using syntax quoting. Syntax quoting usually leads to code that's clearer and more concise because it lets you create a template of the data structure you want to return that's easier to parse visually. Whether you use syntax quoting or plain quoting, it's important to be clear about the distinction between a symbol and the value it evaluates to when building up your list. And if you want your macro to return multiple forms for Clojure to evaluate, make sure to wrap them in a `do`.

Refactoring a Macro and Unquote Splicing

That `code-critic` macro in the preceding section could still use some improvement. Look at the duplication! The two `println` calls are nearly identical. Let's clean that up. First, let's create a function to generate those `println` lists. Functions are easier to think about and play with than macros, so it's often a good idea to move macro guts to helper functions:

```
(defn criticize-code
  [criticism code]
  `(println ~criticism (quote ~code)))

(defmacro code-critic
  [bad good]
  `(do ~(criticize-code "Cursed bacteria of Liberia, this is bad code:" bad)
       ~(criticize-code "Sweet sacred boa of Western and Eastern Samoa, this
is good code:" good)))
```

Notice how the `criticize-code` function returns a syntax-quoted list. This is how you build up the list that the macro will return.

There's more room for improvement, though. The code still has multiple, nearly identical calls to a function. In a situation like this where you want to apply the same function to a collection of values, it makes sense to use a seq function like `map`:

```
(defmacro code-critic
  [bad good]
  `(do ~(map #(apply criticize-code %)
             [["Great squid of Madrid, this is bad code:" bad]
              ["Sweet gorilla of Manila, this is good code:" good]])))
```

This is looking a little better. You're mapping over each criticism/code pair and applying the `criticize-code` function to the pair. Let's try to run the code:

```
(code-critic (1 + 1) (+ 1 1))
; => NullPointerException
```

Oh no! That didn't work at all! What happened? The problem is that `map` returns a list, and in this case, it returned a list of `println` expressions. We just want the result of each `println` call, but instead, this code sticks both results in a list and then tries to evaluate that list.

In other words, as it's evaluating this code, Clojure gets to something like this:

```
(do
 ((clojure.core/println "criticism" '(1 + 1))
  (clojure.core/println "criticism" '(+ 1 1))))
```

then evaluates the first `println` call to give us this:

```
(do
 (nil
  (clojure.core/println "criticism" '(+ 1 1))))
```

and after evaluating the second `println` call, does this:

```
(do
 (nil nil))
```

This is the cause of the exception. `println` evaluates to `nil`, so we end up with something like `(nil nil)`. `nil` isn't callable, and we get a `NullPointerException`.

What an inconvenience! But as it happens, unquote splicing was invented precisely to handle this kind of situation. Unquote splicing is performed with `~@`. If you merely unquote a list, this is what you get:

```
`(+ ~(list 1 2 3))
; => (clojure.core/+ (1 2 3))
```

However, if you use unquote splicing, this is what you get:

```
`(+ ~@(list 1 2 3))
; => (clojure.core/+ 1 2 3)
```

Unquote splicing unwraps a seqable data structure, placing its contents directly within the enclosing syntax-quoted data structure. It's like the `~@` is a sledgehammer and whatever follows it is a piñata, and the result is the most terrifying and awesome party you've ever been to.

Anyway, if you use unquote splicing in your code critic, then everything will work great:

```
(defmacro code-critic
  [{:keys [good bad]}]
  `(do ~@(map #(apply criticize-code %)
              [["Sweet lion of Zion, this is bad code:" bad]
               ["Great cow of Moscow, this is good code:" good]])))
```

```
(code-critic (1 + 1) (+ 1 1))
; => Sweet lion of Zion, this is bad code: (1 + 1)
; => Great cow of Moscow, this is good code: (+ 1 1)
```

Woohoo! You've successfully extracted repetitive code into a function and made your macro code cleaner. Sweet guinea pig of Winnipeg, that is good code!

Things to Watch Out For

Macros have a couple of sneaky gotchas that you should be aware of. In this section, you'll learn about some macro pitfalls and how to avoid them. I hope you haven't unstrapped yourself from your thinking mast.

Variable Capture

Variable capture occurs when a macro introduces a binding that, unknown to the macro's user, eclipses an existing binding. For example, in the following code, a macro mischievously introduces its own `let` binding, and that messes with the code:

```
(def message "Good job!")
(defmacro with-mischief
  [& stuff-to-do]
  (concat (list 'let ['message "Oh, big deal!"])
          stuff-to-do))

(with-mischief
  (println "Here's how I feel about that thing you did: " message))
; => Here's how I feel about that thing you did: Oh, big deal!
```

The `println` call references the symbol `message`, which we think is bound to the string `"Good job!"`. However, the `with-mischief` macro has created a new binding for `message`.

Notice that this macro didn't use syntax quoting. Doing so would result in an exception:

```
(def message "Good job!")
(defmacro with-mischief
  [& stuff-to-do]
  `(let [message "Oh, big deal!"]
     ~@stuff-to-do))

(with-mischief
  (println "Here's how I feel about that thing you did: " message))
; Exception: Can't let qualified name: user/message
```

This exception is for your own good: syntax quoting is designed to prevent you from accidentally capturing variables within macros. If you want to introduce `let` bindings in your macro, you can use a *gensym*. The `gensym` function produces unique symbols on each successive call:

```
(gensym)
; => G__655

(gensym)
; => G__658
```

You can also pass a symbol prefix:

```
(gensym 'message)
; => message4760

(gensym 'message)
; => message4763
```

Here's how you could rewrite `with-mischief` to be less mischievous:

```
(defmacro without-mischief
  [& stuff-to-do]
  (let [macro-message (gensym 'message)]
    `(let [~macro-message "Oh, big deal!"]
       ~@stuff-to-do
       (println "I still need to say: " ~macro-message))))

(without-mischief
  (println "Here's how I feel about that thing you did: " message))
; => Here's how I feel about that thing you did:  Good job!
; => I still need to say:  Oh, big deal!
```

This example avoids variable capture by using `gensym` to create a new, unique symbol that then gets bound to `macro-message`. Within the syntax-quoted `let` expression, `macro-message` is unquoted, resolving to the gensym'd symbol. This gensym'd symbol is distinct from any symbols within `stuff-to-do`, so you avoid variable capture. Because this is such a common pattern, you can use an *auto-gensym*. Auto-gensyms are more concise and convenient ways to use gensyms:

```
`(blarg# blarg#)
(blarg__2869__auto__ blarg__2869__auto__)

`(let [name# "Larry Potter"] name#)
; => (clojure.core/let [name__2872__auto__ "Larry Potter"] name__2872__auto__)
```

In this example, you create an auto-gensym by appending a hash mark (or *hashtag*, if you must insist) to a symbol within a syntax-quoted list. Clojure automatically ensures that each instance of `x#` resolves to the

same symbol within the same syntax-quoted list, that each instance of `y#` resolves similarly, and so on.

`gensym` and auto-gensym are both used all the time when writing macros, and they allow you to avoid variable capture.

Double Evaluation

Another gotcha to watch out for when writing macros is *double evaluation*, which occurs when a form passed to a macro as an argument gets evaluated more than once. Consider the following:

```
(defmacro report
  [to-try]
  `(if ~to-try
     (println (quote ~to-try) "was successful:" ~to-try)
     (println (quote ~to-try) "was not successful:" ~to-try)))

;; Thread/sleep takes a number of milliseconds to sleep for
(report (do (Thread/sleep 1000) (+ 1 1)))
```

This code is meant to test its argument for truthiness. If the argument is truthy, it's considered successful; if it's falsey, it's unsuccessful. The macro prints whether or not its argument was successful. In this case, you would actually sleep for two seconds because `(Thread/sleep 1000)` gets evaluated twice: once right after `if` and again when `println` gets called. This happens because the code `(do (Thread/sleep 1000) (+ 1 1))` is repeated throughout the macro expansion. It's as if you'd written this:

```
(if (do (Thread/sleep 1000) (+ 1 1))
  (println '(do (Thread/sleep 1000) (+ 1 1))
           "was successful:"
           (do (Thread/sleep 1000) (+ 1 1)))

  (println '(do (Thread/sleep 1000) (+ 1 1))
           "was not successful:"
           (do (Thread/sleep 1000) (+ 1 1))))
```

"Big deal!" your inner example critic says. Well, if your code did something like transfer money between bank accounts, this would be a very big deal. Here's how you could avoid this problem:

```
(defmacro report
  [to-try]
  `(let [result# ~to-try]
     (if result#
       (println (quote ~to-try) "was successful:" result#)
       (println (quote ~to-try) "was not successful:" result#))))
```

By placing `to-try` in a `let` expression, you only evaluate that code once and bind the result to an auto-gensym'd symbol, `result#`, which you can now reference without reevaluating the `to-try` code.

Macros All the Way Down

One subtle pitfall of using macros is that you can end up having to write more and more of them to get anything done. This is a consequence of the fact that macro expansion happens before evaluation.

For example, let's say you wanted to `doseq` using the `report` macro. Instead of multiple calls to report:

```
(report (= 1 1))
; => (= 1 1) was successful: true

(report (= 1 2))
; => (= 1 2) was not successful: false
```

let's iterate:

```
(doseq [code ['(= 1 1) '(= 1 2)]]
  (report code))
; => code was successful: (= 1 1)
; => code was successful: (= 1 2)
```

The report macro works fine when we pass it functions individually, but when we use `doseq` to iterate `report` over multiple functions, it's a worthless failure. Here's what a macro expansion for one of the `doseq` iterations would look like:

```
(if
 code
 (clojure.core/println 'code "was successful:" code)
 (clojure.core/println 'code "was not successful:" code))
```

As you can see, `report` receives the unevaluated symbol `code` in each iteration; however, we want it to receive whatever `code` is bound to at evaluation time. But `report`, operating at macro expansion time, just can't access those values. It's like it has T. rex arms, with runtime values forever out of its reach.

To resolve this situation, we might write another macro, like this:

```
(defmacro doseq-macro
  [macroname & args]
  `(do
     ~@(map (fn [arg] (list macroname arg)) args)))

(doseq-macro report (= 1 1) (= 1 2))
; => (= 1 1) was successful: true
; => (= 1 2) was not successful: false
```

If you are ever in this situation, take some time to rethink your approach. It's easy to paint yourself into a corner, making it impossible to accomplish anything with run-of-the-mill function calls. You'll be stuck having to write more macros instead. Macros are extremely powerful and awesome, and you shouldn't be afraid to use them. They turn Clojure's facilities for working with data into facilities for creating new languages informed by your programming problems. For some programs, it's appropriate for your code to be like 90 percent macros. As awesome as they are, they also add new composition challenges. They only really compose with each other, so by using them, you might be missing out on the other kinds of composition (functional, object-oriented) available to you in Clojure.

We've now covered all the mechanics of writing a macro. Pat yourself on the back! It's a pretty big deal!

To close out this chapter, it's finally time to put on your pretending cap and work on that online potion store I talked about at the very beginning of the chapter.

Brews for the Brave and True

At the beginning of this chapter, I revealed a dream: to find some kind of drinkable that, once ingested, would temporarily give me the power and temperament of an '80s fitness guru, freeing me from a prison of inhibition and self-awareness. I'm sure that someone somewhere will someday invent such an elixir, so we might as well get to work on a system for selling this mythical potion. Let's call this hypothetical concoction the *Brave and True Ale*. The name just came to me for no reason whatsoever.

Before the orders come *pouring* in (pun! high-five!), we'll need to have some validation in place. This section shows you a way to do this validation functionally and how to write the code that performs validations a bit more concisely using a macro you'll write called `if-valid`. This will help you understand a typical situation for writing your own macro. If you just want the macro definition, it's okay to skip ahead to "`if-valid`" on page 182.

Validation Functions

To keep things simple, we'll just worry about validating the name and email for each order. For our store, I'm thinking we'll want to have those order details represented like this:

```
(def order-details
  {:name "Mitchard Blimmons"
   :email "mitchard.blimmonsgmail.com"})
```

This particular map has an invalid email address (it's missing the `@` symbol), so this is exactly the kind of order that our validation code should catch! Ideally, we want to write code that produces something like this:

```
(validate order-details order-details-validations)
; => {:email ["Your email address doesn't look like an email address."]}
```

That is, we want to be able to call a function, `validate`, with the data that needs validation and a definition for how to validate it. The result should be a map where each key corresponds to an invalid field, and each value is a vector of one or more validation messages for that field. The following two functions do the job.

Let's look at `order-details-validations` first. Here's how you could represent validations:

```
(def order-details-validations
  {:name
   ["Please enter a name" not-empty]

   :email
   ["Please enter an email address" not-empty

    "Your email address doesn't look like an email address"
    #(or (empty? %) (re-seq #"@" %))]})
```

This is a map where each key is associated with a vector of error message and validating function pairs. For example, `:name` has one validating function, `not-empty`; if that validation fails, you should get the `"Please enter a name"` error message.

Next, we need to write out the `validate` function. The `validate` function can be decomposed into two functions: one to apply validations to a single field and another to accumulate those error messages into a final map of error messages like `{:email ["Your email address doesn't look like an email address."]}`. Here's a function called `error-messages-for` that applies validations to a single value:

```
(defn error-messages-for
  "Return a seq of error messages"
  [to-validate message-validator-pairs]
  (map first (filter #(not ((second %) to-validate))
                     (partition 2 message-validator-pairs))))
```

The first argument, `to-validate`, is the field you want to validate. The second argument, `message-validator-pairs`, should be a seq with an even number of elements. This seq gets grouped into pairs with `(partition 2 message-validator-pairs)`. The first element of the pair should be an error message, and the second element of the pair should be a function (just like

the pairs are arranged in `order-details-validation`). The `error-messages-for` function works by filtering out all error message and validation pairs where the validation function returns `true` when applied to `to-validate`. It then uses `map first` to get the first element of each pair, the error message. Here it is in action:

```
(error-messages-for "" ["Please enter a name" not-empty])
; => ("Please enter a name")
```

Now we need to accumulate these error messages in a map.

Here's the complete `validate` function, as well as the output when we apply it to our `order-details` and `order-details-validations`:

```
(defn validate
  "Returns a map with a vector of errors for each key"
  [to-validate validations]
  (reduce (fn [errors validation]
            (let [[fieldname validation-check-groups] validation
                  value (get to-validate fieldname)
                  error-messages (error-messages-for value validation-check-groups)]
              (if (empty? error-messages)
                errors
                (assoc errors fieldname error-messages))))
          {}
          validations))

(validate order-details order-details-validations)
; => {:email ("Your email address doesn't look like an email address")}
```

Success! This works by reducing over `order-details-validations` and associating the error messages (if there are any) for each key of `order-details` into a final map of error messages.

if-valid

With our validation code in place, we can now validate records to our hearts' content! Most often, validation will look something like this:

```
(let [errors (validate order-details order-details-validations)]
  (if (empty? errors)
    (println :success)
    (println :failure errors)))
```

The pattern is to do the following:

1. Validate a record and bind the result to `errors`
2. Check whether there were any errors
3. If there were, do the success thing, here `(println :success)`
4. Otherwise, do the failure thing, here `(println :failure errors)`

I've actually used this validation code in real production websites. At first, I found myself repeating minor variations of the code over and over, a sure sign that I needed to introduce an abstraction that would hide the repetitive parts: applying the `validate` function, binding the result to some symbol, and checking whether the result is empty. To create this abstraction, you might be tempted to write a function like this:

```
(defn if-valid
  [record validations success-code failure-code]
  (let [errors (validate record validations)]
    (if (empty? errors)
      success-code
      failure-code)))
```

However, this wouldn't work, because `success-code` and `failure-code` would get evaluated each time. A macro would work because macros let you control evaluation. Here's how you'd use the macro:

```
(if-valid order-details order-details-validation errors
 (render :success)
 (render :failure errors))
```

This macro hides the repetitive details and helps you express your intention more succinctly. It's like asking someone to give you the bottle opener instead of saying, "Please give me the manual device for removing the temporary sealant from a glass container of liquid." Here's the implementation:

```
(defmacro if-valid
  "Handle validation more concisely"
  [to-validate validations errors-name & then-else]
  `(let [~errors-name (validate ~to-validate ~validations)]
     (if (empty? ~errors-name)
       ~@then-else)))
```

This macro takes four arguments: `to-validate`, `validations`, `errors-name`, and the rest argument `then-else`. Using `errors-name` like this is a new strategy. We want to have access to the errors returned by the `validate` function within the `then-else` statements. To do this, we tell the macro what symbol it should bind the result to. The following macro expansion shows how this works:

```
 (macroexpand
 '(if-valid order-details order-details-validations my-error-name
            (println :success)
            (println :failure my-error-name)))
(let*
 [my-error-name (user/validate order-details order-details-validations)]
 (if (clojure.core/empty? my-error-name)
  (println :success)
  (println :failure my-error-name)))
```

The syntax quote abstracts the general form of the `let/validate/if` pattern you saw earlier. Then we use unquote splicing to unpack the `if` branches, which were packed into the `then-else` rest argument.

That's pretty simple! After all this talk about macros and going through their mechanics in such detail, I bet you were expecting something more complicated. Sorry, friend. If you're having a hard time coping with your disappointment, I know of a certain drink that will help.

Summary

In this chapter, you learned how to write your own macros. Macros are defined very similarly to functions: they have arguments, a docstring, and a body. They can use argument destructuring and rest args, and they can be recursive. Your macros will almost always return lists. You'll sometimes use `list` and `seq` functions for simple macros, but most of the time you'll use the syntax quote, `, which lets you write macros using a safe template.

When you're writing macros, it's important to keep in mind the distinction between symbols and values: macros are expanded before code is evaluated and therefore don't have access to the results of evaluation. Double evaluation and variable capture are two other subtle traps for the unwary, but you can avoid them through the judicious use of `let` expressions and gensyms.

Macros are fun tools that allow you to code with fewer inhibitions. By letting you control evaluation, macros give you a degree of freedom and expression that other languages simply don't allow. Throughout your Clojure journey, you'll probably hear people cautioning you against their use, saying things like "Macros are evil" and "You should never use macros." Don't listen to these prudes—at least, not at first! Go out there and have a good time. That's the only way you'll learn the situations where it's appropriate to use macros. You'll come out the other side knowing how to use macros with skill and panache.

Exercises

1. Write the macro `when-valid` so that it behaves similarly to `when`. Here is an example of calling it:

```
(when-valid order-details order-details-validation
 (println "It's a success!")
 (render :success))
```

   When the data is valid, the `println` and `render` forms should be evaluated, and `when-valid` should return `nil` if the data is invalid.

2. You saw that `and` is implemented as a macro. Implement `or` as a macro.

3. In Chapter 5 you created a series of functions (`c-int`, `c-str`, `c-dex`) to read an RPG character's attributes. Write a macro that defines an arbitrary number of attribute-retrieving functions using one macro call. Here's how you would call it:

```
(defattrs c-int :intelligence
          c-str :strength
          c-dex :dexterity)
```

PART III

ADVANCED TOPICS

9

THE SACRED ART OF CONCURRENT AND PARALLEL PROGRAMMING

If I were the lord of a manor and you were my heir, I would sit you down on your 13th name day and tell you, "The world of computing is changing, lass, and ye must be prepared for the new world of multi-core processors lest ye be trampled by it.

"Listen well: In recent years, CPU clock speeds have barely increased, but dual-core and quad-core computers have become common. The laws of physics are cruel and absolute, and they demand that increasing clock speed requires exponentially more power. The realm's best engineers are unlikely to overcome this limitation anytime soon, if ever. Therefore, you can expect the trend of increasing cores on a single machine to continue—as will the expectation that you as a programmer will know how to make the most of modern hardware.

"Learning to program in this new paradigm will be fun and fascinating, verily. But beware: it is also fraught with peril. You must learn *concurrent and parallel programming*, which is the sacred art of structuring your application to safely manage multiple, simultaneously executing tasks.

"You begin your instruction in this art with an overview of concurrency and parallelism concepts. You'll then study the three goblins that harry every practitioner: reference cells, mutual exclusion, and dwarven berserkers. And you'll learn three tools that will aid you: futures, promises, and delays."

And then I'd tap you on the shoulder with a keyboard, signaling that you were ready to begin.

Concurrency and Parallelism Concepts

Concurrent and parallel programming involves a lot of messy details at all levels of program execution, from the hardware to the operating system to programming language libraries to the code that springs from your heart and lands in your editor. But before you worry your head with any of those details, in this section I'll walk through the high-level concepts that surround concurrency and parallelism.

Managing Multiple Tasks vs. Executing Tasks Simultaneously

Concurrency refers to managing more than one task at the same time. *Task* just means "something that needs to get done," and it doesn't imply anything regarding implementation in your hardware or software. We can illustrate concurrency with the song "Telephone" by Lady Gaga. Gaga sings,

> I cannot text you with a drink in my hand, eh

Here, she's explaining that she can only manage one task (drinking). She flat-out rejects the suggestion that she can manage more than one task. However, if she decided to process tasks concurrently, she would sing,

> I will put down this drink to text you, then put my phone away
> and continue drinking, eh

In this hypothetical universe, Lady Gaga is managing two tasks: drinking and texting. However, she is not executing both tasks at the same time. Instead, she's switching between the two, or *interleaving*. Note that, while interleaving, you don't have to fully complete a task before switching: Gaga could type one word, put down her phone, pick up her drink and take a sip, and then switch back to her phone and type another word.

Parallelism refers to executing more than one task at the same time. If Madame Gaga were to execute her two tasks in parallel, she would sing,

> I can text you with one hand while I use the other to drink, eh

Parallelism is a subclass of concurrency: before you execute multiple tasks simultaneously, you first have to manage multiple tasks.

Clojure has many features that allow you to achieve parallelism easily. While the Lady Gaga system achieves parallelism by simultaneously executing tasks on multiple hands, computer systems generally achieve parallelism by simultaneously executing tasks on multiple processors.

It's important to distinguish parallelism from *distribution*. Distributed computing is a special version of parallel computing where the processors are in different computers and tasks are distributed to computers over a network. It'd be like Lady Gaga asking Beyoncé, "Please text this guy while I drink." Although you can do distributed programming in Clojure with the aid of libraries, this book covers only parallel programming, and here I'll use *parallel* to refer only to cohabiting processors. If you're interested in distributed programming, check out Kyle Kingsbury's *Call Me Maybe* series at *https://aphyr.com/*.

Blocking and Asynchronous Tasks

One of the major use cases for concurrent programming is for *blocking* operations. Blocking really just means waiting for an operation to finish. You'll most often hear it used in relation to I/O operations, like reading a file or waiting for an HTTP request to finish. Let's examine this using the concurrent Lady Gaga example.

If Lady Gaga texts her interlocutor and then stands there with her phone in her hand, staring at the screen for a response and not drinking, then you would say that the *read next text message* operation is blocking and that these tasks are executing *synchronously*.

If, instead, she tucks her phone away so she can drink until it alerts her by beeping or vibrating, then the *read next text message* task is not blocking and you would say she's handling the task *asynchronously*.

Concurrent Programming and Parallel Programming

Concurrent programming and parallel programming refer to techniques for decomposing a task into subtasks that can execute in parallel and managing the risks that arise when your program executes more than one task at the same time. For the rest of the chapter, I'll use the two terms interchangeably because the risks are pretty much the same for both.

To better understand those risks and how Clojure helps you avoid them, let's examine how concurrency and parallelism are implemented in Clojure.

Clojure Implementation: JVM Threads

I've been using the term *task* in an abstract sense to refer to a series of related operations without regard for how a computer might implement the task concept. For example, texting is a task that consists of a series of related operations that are totally separate from the operations involved in pouring a drink into your face.

In Clojure, you can think of your normal, *serial* code as a sequence of tasks. You indicate that tasks can be performed concurrently by placing them on JVM *threads*.

What's a Thread?

I'm glad you asked! A thread is a subprogram. A program can have many threads, and each thread executes its own set of instructions while enjoying shared access to the program's state.

Thread management functionality can exist at multiple levels in a computer. For example, the operating system kernel typically provides system calls to create and manage threads. The JVM provides its own platform-independent thread management functionality, and since Clojure programs run in the JVM, they use JVM threads. You'll learn more about the JVM in Chapter 12.

You can think of a thread as an actual, physical piece of thread that strings together a sequence of instructions. In my mind, the instructions are marshmallows, because marshmallows are delicious. The processor executes these instructions in order. I picture this as an alligator consuming the instructions, because alligators love marshmallows (true fact!). So executing a program looks like a bunch of marshmallows strung out on a line with an alligator traveling down the line and eating them one by one. Figure 9-1 shows this model for a single-core processor executing a single-threaded program.

Figure 9-1: Single-core processor executing a single-threaded program

A thread can *spawn* a new thread to execute tasks concurrently. In a single-processor system, the processor switches back and forth between the threads (interleaving). Here's where potential concurrency issues get introduced. Although the processor executes the instructions on each thread in order, it makes no guarantees about when it will switch back and forth between threads.

Figure 9-2 shows an illustration of two threads, A and B, and a timeline of how their instructions could be executed. I've shaded the instructions on thread B to help distinguish them from the instructions on thread A.

Figure 9-2: Single-core processor executing two threads

Note that this is just one possible order of instruction execution. The processor could also have executed the instructions in the order A1, A2, A3, B1, A4, B2, B3 for example. This makes the program *nondeterministic*. You can't know beforehand what the result will be because you can't know the execution order, and different execution orders can yield different results.

This example shows concurrent execution on a single processor through interleaving, whereas a multi-core system assigns a thread to each core, allowing the computer to execute more than one thread simultaneously. Each core executes its thread's instructions in order, as shown in Figure 9-3.

Figure 9-3: Two threads, two processors

As with interleaving on a single core, there are no guarantees for the overall execution order, so the program is nondeterministic. When you add a second thread to a program, it becomes nondeterministic, and this makes it possible for your program to fall prey to three kinds of problems.

The Three Goblins: Reference Cells, Mutual Exclusion, and Dwarven Berserkers

There are three central challenges in concurrent programming, also known as the *The Three Concurrency Goblins*. To see why these are scary, imagine that the program in the image in Figure 9-3 includes the pseudo-instructions in Table 9-1.

Table 9-1: Instructions for a Program with a Nondeterministic Outcome

ID	Instruction
A1	WRITE X = 0
A2	READ X
A3	WRITE X = X + 1
B1	READ X
B2	WRITE X = X + 1

If the processor follows the order A1, A2, A3, B1, B2, then X will have a value of 2, as you'd expect. But if it follows the order A1, A2, B1, A3, B2, X's value will be 1, as you can see in Figure 9-4.

Figure 9-4: Two threads interacting with a reference cell

We'll call this the *reference cell* problem (the first Concurrency Goblin). The reference cell problem occurs when two threads can read and write to the same location, and the value at the location depends on the order of the reads and writes.

The second Concurrency Goblin is *mutual exclusion*. Imagine two threads, each trying to write a spell to a file. Without any way to claim exclusive write access to the file, the spell will end up garbled because the write instructions will be interleaved. Consider the following two spells:

> By the power invested in me
> by the state of California,
> I now pronounce you man and wife
>
> Thunder, lightning, wind, and rain,
> a delicious sandwich, I summon again

If you write these to a file without mutual exclusion, you could end up with this:

> By the power invested in me
> by Thunder, lightning, wind, and rain,

the state of California,
I now pronounce you a delicious man sandwich, and wife
I summon again

The third Concurrency Goblin is what I'll call the *dwarven berserker* problem (aka *deadlock*). Imagine four berserkers sitting around a rough-hewn, circular wooden table comforting each other. "I know I'm distant toward my children, but I just don't know how to communicate with them," one growls. The rest sip their coffee and nod knowingly, care lines creasing their eye places.

Now, as everyone knows, the dwarven berserker ritual for ending a comforting coffee klatch is to pick up their "comfort sticks" (double-bladed war axes) and scratch each other's backs. One war axe is placed between each pair of dwarves, as shown in Figure 9-5.

Their ritual proceeds thusly:

1. Pick up the *left* war axe, when available.
2. Pick up the *right* war axe, when available.
3. Comfort your neighbor with vigorous swings of your "comfort sticks."
4. Release both war axes.
5. Repeat.

Figure 9-5: Dwarven berserkers at a comforting coffee klatch

Following this ritual, it's entirely possible that all the dwarven berserkers will pick up their left comfort stick and then block indefinitely while waiting for the comfort stick to their right to become available, resulting in deadlock. (By the way, if you want to look into this phenomenon further, it's

usually referred to as the *dining philosophers problem*, but that's a more boring scenario.) This book doesn't discuss deadlock in much detail, but it's good to know the concept and its terminology.

Concurrent programming has its goblins, but with the right tools, it's manageable and even fun. Let's start looking at the right tools.

Futures, Delays, and Promises

Futures, delays, and promises are easy, lightweight tools for concurrent programming. In this section, you'll learn how each one works and how to use them together to defend against the reference cell Concurrency Goblin and the mutual exclusion Concurrency Goblin. You'll discover that, although simple, these tools go a long way toward meeting your concurrency needs.

They do this by giving you more flexibility than is possible with serial code. When you write serial code, you bind together these three events:

- Task definition
- Task execution
- Requiring the task's result

As an example, look at this hypothetical code, which defines a simple API call task:

```
(web-api/get :dwarven-beard-waxes)
```

As soon as Clojure encounters this task definition, it executes it. It also requires the result *right now*, blocking until the API call finishes. Part of learning concurrent programming is learning to identify when these chronological couplings aren't necessary. Futures, delays, and promises allow you to separate task definition, task execution, and requiring the result. Onward!

Futures

In Clojure, you can use *futures* to define a task and place it on another thread without requiring the result immediately. You can create a future with the `future` macro. Try this in a REPL:

```
(future (Thread/sleep 4000)
        (println "I'll print after 4 seconds"))
(println "I'll print immediately")
```

`Thread/sleep` tells the current thread to just sit on its bum and do nothing for the specified number of milliseconds. Normally, if you evaluated `Thread/sleep` in your REPL, you wouldn't be able to evaluate any other statements until the REPL was done sleeping; the thread executing your REPL

would be blocked. However, future creates a new thread and places each expression you pass it on the new thread, including Thread/sleep, allowing the REPL's thread to continue, unblocked.

You can use futures to run tasks on a separate thread and then forget about them, but often you'll want to use the result of the task. The future function returns a reference value that you can use to request the result. The reference is like the ticket that a dry cleaner gives you: at any time you can use it to request your clean dress, but if your dress isn't clean yet, you'll have to wait. Similarly, you can use the reference value to request a future's result, but if the future isn't done computing the result, you'll have to wait.

Requesting a future's result is called *dereferencing* the future, and you do it with either the deref function or the @ reader macro. A future's result value is the value of the last expression evaluated in its body. A future's body executes only once, and its value gets cached. Try the following:

```
(let [result (future (println "this prints once")
                     (+ 1 1))]
  (println "deref: " (deref result))
  (println "@: " @result))
; => "this prints once"
; => deref: 2
; => @: 2
```

Notice that the string "this prints once" indeed prints only once, even though you dereference the future twice. This shows that the future's body ran only once and the result, 2, got cached.

Dereferencing a future will block if the future hasn't finished running, like so:

```
(let [result (future (Thread/sleep 3000)
                     (+ 1 1))]
  (println "The result is: " @result)
  (println "It will be at least 3 seconds before I print"))
; => The result is: 2
; => It will be at least 3 seconds before I print
```

Sometimes you want to place a time limit on how long to wait for a future. To do that, you can pass deref a number of milliseconds to wait along with the value to return if the deref times out:

```
(deref (future (Thread/sleep 1000) 0) 10 5)
; => 5
```

This code tells deref to return the value 5 if the future doesn't return a value within 10 milliseconds.

Finally, you can interrogate a future using `realized?` to see if it's done running:

```
(realized? (future (Thread/sleep 1000)))
; => false

(let [f (future)]
  @f
  (realized? f))
; => true
```

Futures are a dead-simple way to sprinkle some concurrency on your program.

On their own, they give you the power to chuck tasks onto other threads, which can make your program more efficient. They also let your program behave more flexibly by giving you control over when a task's result is required.

When you dereference a future, you indicate that the result is required *right now* and that evaluation should stop until the result is obtained. You'll see how this can help you deal with the mutual exclusion problem in just a bit. Alternatively, you can ignore the result. For example, you can use futures to write to a log file asynchronously, in which case you don't need to dereference the future to get any value back.

The flexibility that futures give you is very cool. Clojure also allows you to treat task definition and requiring the result independently with delays and promises.

Delays

Delays allow you to define a task without having to execute it or require the result immediately. You can create a delay using `delay`:

```
(def jackson-5-delay
  (delay (let [message "Just call my name and I'll be there"]
           (println "First deref:" message)
           message)))
```

In this example, nothing is printed, because we haven't yet asked the `let` form to be evaluated. You can evaluate the delay and get its result by dereferencing it or by using `force`. `force` behaves identically to `deref` in that it communicates more clearly that you're causing a task to start as opposed to waiting for a task to finish:

```
(force jackson-5-delay)
; => First deref: Just call my name and I'll be there
; => "Just call my name and I'll be there"
```

Like futures, a delay is run only once and its result is cached. Subsequent dereferencing will return the Jackson 5 message without printing anything:

```
@jackson-5-delay
; => "Just call my name and I'll be there"
```

One way you can use a delay is to fire off a statement the first time one future out of a group of related futures finishes. For example, pretend your app uploads a set of headshots to a headshot-sharing site and notifies the owner as soon as the first one is up, as in the following:

```
(def gimli-headshots ["serious.jpg" "fun.jpg" "playful.jpg"])
(defn email-user
  [email-address]
  (println "Sending headshot notification to" email-address))
(defn upload-document
  "Needs to be implemented"
  [headshot]
  true)
(let [notify (delay ❶(email-user "and-my-axe@gmail.com"))]
  (doseq [headshot gimli-headshots]
    (future (upload-document headshot)
            ❷(force notify))))
```

In this example, you define a vector of headshots to upload (`gimli-headshots`) and two functions (`email-user` and `upload-document`) to pretend-perform the two operations. Then you use `let` to bind `notify` to a delay. The body of the delay, (`email-user "and-my-axe@gmail.com"`) ❶, isn't evaluated when the delay is created. Instead, it gets evaluated the first time one of the futures created by the `doseq` form evaluates (`force notify`) ❷. Even though (`force notify`) will be evaluated three times, the delay body is evaluated only once. Gimli will be grateful to know when the first headshot is available so he can begin tweaking it and sharing it. He'll also appreciate not being spammed, and you'll appreciate not facing his dwarven wrath.

This technique can help protect you from the mutual exclusion Concurrency Goblin—the problem of making sure that only one thread can access a particular resource at a time. In this example, the delay guards the email server resource. Because the body of a delay is guaranteed to fire only once, you can be sure that you will never run into a situation where two threads send the same email. Of course, no thread will ever be able to use the delay to send an email again. That might be too drastic a constraint for most situations, but in cases like this example, it works perfectly.

Promises

Promises allow you to express that you expect a result without having to define the task that should produce it or when that task should run. You create promises using `promise` and deliver a result to them using `deliver`. You obtain the result by dereferencing:

```
(def my-promise (promise))
(deliver my-promise (+ 1 2))
@my-promise
; => 3
```

Here, you create a promise and then deliver a value to it. Finally, you obtain the value by dereferencing the promise. Dereferencing is how you express that you expect a result, and if you had tried to dereference `my-promise` without first delivering a value, the program would block until a promise was delivered, just like with futures and delays. You can only deliver a result to a promise once.

One use for promises is to find the first satisfactory element in a collection of data. Suppose, for example, that you're gathering ingredients to make your parrot sound like James Earl Jones. Because James Earl Jones has the smoothest voice on earth, one of the ingredients is premium yak butter with a smoothness rating of 97 or greater. You have a budget of $100 for one pound.

You are a modern practitioner of the magico-ornithological arts, so rather than tediously navigating each yak butter retail site, you create a script to give you the URL of the first yak butter that meets your needs.

The following code defines some yak butter products, creates a function to mock up an API call, and creates another function to test whether a product is satisfactory:

```
(def yak-butter-international
  {:store "Yak Butter International"
    :price 90
    :smoothness 90})
(def butter-than-nothing
  {:store "Butter Than Nothing"
   :price 150
   :smoothness 83})
```

```
;; This is the butter that meets our requirements
(def baby-got-yak
  {:store "Baby Got Yak"
   :price 94
   :smoothness 99})

(defn mock-api-call
  [result]
  (Thread/sleep 1000)
  result)

(defn satisfactory?
  "If the butter meets our criteria, return the butter, else return false"
  [butter]
  (and (<= (:price butter) 100)
       (>= (:smoothness butter) 97)
       butter))
```

The API call waits one second before returning a result to simulate the time it would take to perform an actual call.

To show how long it will take to check the sites synchronously, we'll use `some` to apply the `satisfactory?` function to each element of the collection and return the first truthy result, or nil if there are none. When you check each site synchronously, it could take more than one second per site to obtain a result, as the following code shows:

```
(time (some (comp satisfactory? mock-api-call)
            [yak-butter-international butter-than-nothing baby-got-yak]))
; => "Elapsed time: 3002.132 msecs"
; => {:store "Baby Got Yak", :smoothness 99, :price 94}
```

Here I've used `comp` to compose functions, and I've used `time` to print the time taken to evaluate a form. You can use a promise and futures to perform each check on a separate thread. If your computer has multiple cores, this could reduce the time it takes to about one second:

```
(time
 (let [butter-promise (promise)]
   (doseq [butter [yak-butter-international butter-than-nothing baby-got-yak]]
     (future (if-let [satisfactory-butter (satisfactory? (mock-api-call butter))]
               (deliver butter-promise satisfactory-butter))))
   (println "And the winner is:" @butter-promise)))
; => "Elapsed time: 1002.652 msecs"
; => And the winner is: {:store Baby Got Yak, :smoothness 99, :price 94}
```

In this example, you first create a promise, `@butter-promise`, and then create three futures with access to that promise. Each future's task is to evaluate a yak butter site and to deliver the site's data to the promise if it's satisfactory. Finally, you dereference `@butter-promise`, causing the program to block until the site data is delivered. This takes about one second instead

of three because the site evaluations happen in parallel. By decoupling the requirement for a result from how the result is actually computed, you can perform multiple computations in parallel and save some time.

You can view this as a way to protect yourself from the reference cell Concurrency Goblin. Because promises can be written to only once, you prevent the kind of inconsistent state that arises from nondeterministic reads and writes.

You might be wondering what happens if none of the yak butter is satisfactory. If that happens, the dereference would block forever and tie up the thread. To avoid that, you can include a timeout:

```
(let [p (promise)]
  (deref p 100 "timed out"))
```

This creates a promise, `p`, and tries to dereference it. The number 100 tells `deref` to wait 100 milliseconds, and if no value is available by then, to use the timeout value, `"timed out"`.

The last detail I should mention is that you can also use promises to register callbacks, achieving the same functionality that you might be used to in JavaScript. JavaScript callbacks are a way of defining code that should execute asynchronously once some other code finishes. Here's how to do it in Clojure:

```
(let [ferengi-wisdom-promise (promise)]
  (future (println "Here's some Ferengi wisdom:" @ferengi-wisdom-promise))
  (Thread/sleep 100)
  (deliver ferengi-wisdom-promise "Whisper your way to success."))
; => Here's some Ferengi wisdom: Whisper your way to success.
```

This example creates a future that begins executing immediately. However, the future's thread is blocking because it's waiting for a value to be delivered to `ferengi-wisdom-promise`. After 100 milliseconds, you deliver the value and the `println` statement in the future runs.

Futures, delays, and promises are great, simple ways to manage concurrency in your application. In the next section, we'll look at one more fun way to keep your concurrent applications under control.

Rolling Your Own Queue

So far you've looked at some simple ways to combine futures, delays, and promises to make your concurrent programs a little safer. In this section, you'll use a macro to combine futures and promises in a slightly more complex manner. You might not necessarily ever use this code, but it'll show the power of these modest tools a bit more. The macro will require you to hold runtime logic and macro expansion logic in your head at the same time to understand what's going on; if you get stuck, just skip ahead.

One characteristic The Three Concurrency Goblins have in common is that they all involve tasks concurrently accessing a shared resource—a variable, a printer, a dwarven war axe—in an uncoordinated way. If you want to ensure that only one task will access a resource at a time, you can place the resource access portion of a task on a queue that's executed serially. It's kind of like making a cake: you and a friend can separately retrieve the ingredients (eggs, flour, eye of newt, what have you), but some steps you'll have to perform serially. You have to prepare the batter before you put it in the oven. Figure 9-6 illustrates this strategy.

Figure 9-6: Dividing tasks into a serial portion and a concurrent portion lets you safely make your code more efficient.

To implement the queuing macro, you'll pay homage to the British, because they invented queues. You'll use a queue to ensure that the customary British greeting "Ello, gov'na! Pip pip! Cheerio!" is delivered in the correct order. This demonstration will involve an abundance of `sleeping`, so here's a macro to do that more concisely:

```
(defmacro wait
  "Sleep `timeout` seconds before evaluating body"
  [timeout & body]
  `(do (Thread/sleep ~timeout) ~@body))
```

All this code does is take whatever forms you give it and insert a call to `Thread/sleep` before them, all wrapped up in `do`.

The code in Listing 9-1 splits up tasks into a concurrent portion and a serialized portion:

```
(let [saying3 (promise)]
  (future (deliver saying3 (wait 100 "Cheerio!")))
  @(let [saying2 (promise)]
     (future (deliver saying2 (wait 400 "Pip pip!")))
❶    @(let [saying1 (promise)]
        (future (deliver saying1 (wait 200 "'Ello, gov'na!")))
```

```
        (println @saying1)
        saying1)
      (println @saying2)
      saying2)
  (println @saying3)
  saying3)
```

Listing 9-1: The expansion of an enqueue macro call

The overall strategy is to create a promise for each task (in this case, printing part of the greeting) to create a corresponding future that will deliver a concurrently computed value to the promise. This ensures that all of the futures are created before any of the promises are dereferenced, and it ensures that the serialized portions are executed in the correct order. The value of `saying1` is printed first—`"'Ello, gov'na!"`—then the value of `saying2`, and finally `saying3`. Returning `saying1` in a `let` block and dereferencing the `let` block at ❶ ensures that you'll be completely finished with `saying1` before the code moves on to do anything to `saying2`, and this pattern is repeated with `saying2` and `saying3`.

It might seem silly to dereference the `let` block, but doing so lets you abstract this code with a macro. And you will definitely want to use a macro, because writing out code like the previous example would drive you mental (as the British would say). Ideally, the macro would work as shown in Listing 9-2:

```
(-> (enqueue ❶saying ❷(wait 200 "'Ello, gov'na!") ❸(println @saying))
  ❹(enqueue saying (wait 400 "Pip pip!") (println @saying))
    (enqueue saying (wait 100 "Cheerio!") (println @saying)))
```

Listing 9-2: This is how you'd use enqueue.

The macro lets you name the promise that gets created ❶, define how to derive the value to deliver that promise ❷, and define what to do with the promise ❸. The macro can also take another `enqueue` macro call as its first argument, which lets you thread it ❹. Listing 9-3 shows how you can define the `enqueue` macro. After defining `enqueue`, the code in Listing 9-2 will expand into the code in Listing 9-1, with all the nested `let` expressions:

```
  (defmacro enqueue
❶   ([q concurrent-promise-name concurrent serialized]
❷    `(let [~concurrent-promise-name (promise)]
        (future (deliver ~concurrent-promise-name ~concurrent))
❸       (deref ~q)
        ~serialized
        ~concurrent-promise-name))
❹   ([concurrent-promise-name concurrent serialized]
     `(enqueue (future) ~concurrent-promise-name ~concurrent ~serialized)))
```

Listing 9-3: enqueue's implementation

Notice first that this macro has two arities in order to supply a default value. The first arity ❶ is where the real work is done. It has the parameter `q`, and the second arity does not. The second arity ❹ calls the first with value (`future`) supplied for `q`; you'll see why in a minute. At ❷, the macro returns a form that creates a promise, delivers its value in a future, dereferences whatever form is supplied for `q`, evaluates the serialized code, and finally returns the promise. `q` will usually be a nested `let` expression returned by another call to `enqueue`, like in Listing 9-2. If no value is supplied for `q`, the macro supplies a future so that the `deref` at ❸ doesn't cause an exception.

Now that we've written the `enqueue` macro, let's try it out to see whether it reduces the execution time!

```
(time @(-> (enqueue saying (wait 200 "'Ello, gov'na!") (println @saying))
           (enqueue saying (wait 400 "Pip pip!") (println @saying))
           (enqueue saying (wait 100 "Cheerio!") (println @saying))))
; => 'Ello, gov'na!
; => Pip pip!
; => Cheerio!
; => "Elapsed time: 401.635 msecs"
```

Blimey! The greeting is delivered in the correct order, and you can see by the elapsed time that the "work" of sleeping was handled concurrently.

Summary

It's important for programmers like you to learn concurrent and parallel programming techniques so you can design programs that run efficiently on modern hardware. Concurrency refers to a program's ability to carry out more than one task, and in Clojure you achieve this by placing tasks on separate threads. Programs execute in parallel when a computer has more than one CPU, which allows more than one thread to be executed at the same time.

Concurrent programming refers to the techniques used to manage three concurrency risks: reference cells, mutual exclusion, and deadlock. Clojure gives you three basic tools that help you mitigate those risks: futures, delays, and promises. Each tool lets you decouple the three events of defining a task, executing a task, and requiring a task's result. Futures let you define a task and execute it immediately, allowing you to require the result later or never. Futures also cache their results. Delays let you define a task that doesn't get executed until later, and a delay's result gets cached. Promises let you express that you require a result without having to know about the task that produces that result. You can only deliver a value to a promise once.

In the next chapter, you'll explore the philosophical side of concurrent programming and learn more sophisticated tools for managing the risks.

Exercises

1. Write a function that takes a string as an argument and searches for it on Bing and Google using the `slurp` function. Your function should return the HTML of the first page returned by the search.
2. Update your function so it takes a second argument consisting of the search engines to use.
3. Create a new function that takes a search term and search engines as arguments, and returns a vector of the URLs from the first page of search results from each search engine.

10

CLOJURE METAPHYSICS: ATOMS, REFS, VARS, AND CUDDLE ZOMBIES

The Three Concurrency Goblins are all spawned from the same pit of evil: shared access to mutable state. You can see this in the reference cell discussion in Chapter 9. When two threads make uncoordinated changes to the reference cell, the result is unpredictable.

Rich Hickey designed Clojure to specifically address the problems that develop from shared access to mutable state. In fact, Clojure embodies a very clear conception of state that makes it inherently safer for concurrency than most popular programming languages. It's safe all the way down to its *meta-freakin-physics*.

In this chapter, you'll learn about Clojure's underlying metaphysics, as compared to the metaphysics of typical object-oriented (OO) languages. Learning this philosophy will prepare you to handle Clojure's remaining concurrency tools, the *atom*, *ref*, and *var* reference types. (Clojure has one additional reference type, *agents*, which this book doesn't cover.) Each of

these types enables you to safely perform state-modifying operations concurrently. You'll also learn about easy ways to make your program more efficient without introducing state at all.

Metaphysics attempts to answer two basic questions in the broadest possible terms:

- What is there?
- What is it like?

To draw out the differences between Clojure and OO languages, I'll explain two different ways of modeling a cuddle zombie. Unlike a regular zombie, a cuddle zombie does not want to devour your brains. It only wants to spoon you and maybe smell your neck. That makes its undead, shuffling, decaying state all the more tragic. How could you try to kill something that only wants love? Who's the real monster here?

Object-Oriented Metaphysics

OO metaphysics treats the cuddle zombie as an object that exists in the world. The object has properties that may change over time, but it's still treated as a single, constant object. If that seems like a totally obvious, uncontroversial approach to zombie metaphysics, you probably haven't spent hours in an intro philosophy class arguing about what it means for a chair to exist and what really makes it a chair in the first place.

The tricky part is that the cuddle zombie is always changing. Its body slowly deteriorates. Its undying hunger for cuddles grows fiercer with time. In OO terms, we would say that the cuddle zombie is an object with mutable state and that its state is ever fluctuating. But no matter how much the zombie changes, we still identify it as the same zombie. Here's how you might model and interact with a cuddle zombie in Ruby:

```
class CuddleZombie
  # attr_accessor is just a shorthand way for creating getters and
  # setters for the listed instance variables
  attr_accessor :cuddle_hunger_level, :percent_deteriorated

  def initialize(cuddle_hunger_level = 1, percent_deteriorated = 0)
    self.cuddle_hunger_level = cuddle_hunger_level
    self.percent_deteriorated = percent_deteriorated
  end
end

fred = CuddleZombie.new(2, 3)
fred.cuddle_hunger_level  # => 2
fred.percent_deteriorated # => 3

fred.cuddle_hunger_level = 3
fred.cuddle_hunger_level # => 3
```

In this example, you create a cuddle zombie, `fred`, with two attributes: `cuddle_hunger_level` and `percent_deteriorated`. `fred` starts out with a `cuddle_hunger_level` of just 2, but you can change it to whatever you want and it's still good ol' Fred, the same cuddle zombie. In this case, you changed its `cuddle_hunger_level` to 3.

You can see that this object is just a fancy reference cell. It's subject to the same nondeterministic results in a multithreaded environment. For example, if two threads try to increment Fred's hunger level with something like `fred.cuddle_hunger_level = fred.cuddle_hunger_level + 1`, one of the increments could be lost, just like in the example with two threads writing to `X` in "The Three Goblins: Reference Cells, Mutual Exclusion, and Dwarven Berserkers" on page 193.

Even if you're only performing reads on a separate thread, the program will still be nondeterministic. For example, suppose you're conducting research on cuddle zombie behavior. You want to log a zombie's hunger level whenever it reaches 50 percent deterioration, but you want to do this on another thread to increase performance, using code like that in Listing 10-1:

```
if fred.percent_deteriorated >= 50
  Thread.new { database_logger.log(fred.cuddle_hunger_level) }
end
```

Listing 10-1: This Ruby code isn't safe for concurrent execution.

The problem is that another thread could change `fred` before the write actually takes place.

For example, Figure 10-1 shows two threads executing from top to bottom. In this situation, it would be correct to write 5 to the database, but 10 gets written instead.

Figure 10-1: Logging inconsistent cuddle zombie data

This would be unfortunate. You don't want your data to be inconsistent when you're trying to recover from the cuddle zombie apocalypse. However, there's no way to retain the state of an object at a specific moment in time.

Additionally, in order to change the `cuddle_hunger_level` and `percent_deteriorated` simultaneously, you must be extra careful. Otherwise, it's possible for `fred` to be viewed in an inconsistent state, because another thread might `read` the `fred` object in between the two changes that you intend to be simultaneous, like so:

```
fred.cuddle_hunger_level = fred.cuddle_hunger_level + 1
# At this time, another thread could read fred's attributes and
# "perceive" fred in an inconsistent state unless you use a mutex
fred.percent_deteriorated = fred.percent_deteriorated + 1
```

This is another version of the mutual exclusion problem. In object-oriented programming (OOP), you can manually address this problem with a *mutex*, which ensures that only one thread can access a resource (in this case, the `fred` object) at a time for the duration of the mutex.

The fact that objects are never stable doesn't stop us from treating them as the fundamental building blocks of programs. In fact, this is considered an advantage of OOP. It doesn't matter how the state changes; you can still interact with a stable interface and everything will work as it should. This conforms to our intuitive sense of the world. A piece of wax is still the same piece of wax even if its properties change: if I change its color, melt it, and pour it on the face of my enemy, I'd still think of it as the same wax object I started with.

Also, in OOP, objects do things. They act on each other, changing state as the program runs. Again, this conforms to our intuitive sense of the world: change is the result of objects acting on each other. A Person object pushes on a Door object and enters a House object.

Clojure Metaphysics

In Clojure metaphysics, we would say that we never encounter the same cuddle zombie twice. The cuddle zombie is not a discrete thing that exists in the world independent of its mutations: it's actually a succession of *values*.

The term *value* is used often by Clojurists, and its specific meaning might differ from what you're used to. Values are *atomic* in the sense that they form a single irreducible unit or component in a larger system; they're indivisible, unchanging, stable entities. Numbers are values: it wouldn't make sense for the number 15 to mutate into another number. When you add or subtract from 15, you don't change the number 15; you just wind up with a different number. Clojure's data structures are also values because they're immutable. When you use `assoc` on a map, you don't modify the original map; instead, you derive a new map.

So a value doesn't change, but you can apply a *process* to a value to produce a new value. For example, say we start with a value *F1*, and then we apply the *Cuddle Zombie* process to *F1* to produce the value *F2*. The process then gets applied to the value *F2* to produce the value *F3*, and so on.

This leads to a different conception of *identity*. Instead of understanding identity as inherent to a changing object, as in OO metaphysics, Clojure metaphysics construes identity as something we humans impose on a succession of unchanging values produced by a process over time. We use *names* to designate identities. The name *Fred* is a handy way to refer to a series of individual states *F1*, *F2*, *F3*, and so on. From this viewpoint, there's no such thing as mutable state. Instead, *state* means the value of an identity at a point in time.

Rich Hickey has used the analogy of phone numbers to explain state. *Alan's phone number* has changed 10 times, but we will always call these numbers by the same name, *Alan's phone number*. Alan's phone number five years ago is a different value than Alan's phone number today, and both are two states of Alan's phone number identity.

This makes sense when you consider that in your programs you are dealing with information about the world. Rather than saying that information has changed, you would say you've received new information. At 12:00 PM on Friday, Fred the Cuddle Zombie was in a state of 50 percent decay. At 1:00 PM, he was 60 percent decayed. These are both facts that you can process, and the introduction of a new fact does not invalidate a previous fact. Even though Fred's decay increased from 50 percent to 60 percent, it's still true that at 12:00 PM he was in a state of 50 percent decay.

Figure 10-2 shows how you might visualize values, process, identity, and state.

Figure 10-2: Values, process, identity, and state

These values don't act on each other, and they can't be changed. They can't *do* anything. Change only occurs when a) a process generates a new value and b) we choose to associate the identity with the new value.

To handle this sort of change, Clojure uses *reference types*. Reference types let you manage identities in Clojure. Using them, you can name an identity and retrieve its state. Let's look at the simplest of these, the *atom*.

Atoms

Clojure's atom reference type allows you to endow a succession of related values with an identity. Here's how you create one:

```
(def fred (atom {:cuddle-hunger-level 0
                 :percent-deteriorated 0}))
```

This creates a new atom and binds it to the name `fred`. This atom *refers* to the value `{:cuddle-hunger-level 0 :percent-deteriorated 0}`, and you would say that that's its current state.

To get an atom's current state, you dereference it. Here's Fred's current state:

```
@fred
; => {:cuddle-hunger-level 0, :percent-deteriorated 0}
```

Unlike futures, delays, and promises, dereferencing an atom (or any other reference type) will never block. When you dereference futures, delays, and promises, it's like you're saying "I need a value now, and I will wait until I get it," so it makes sense that the operation would block. However, when you dereference a reference type, it's like you're saying "give me the value I'm currently referring to," so it makes sense that the operation doesn't block, because it doesn't have to wait for anything.

In the Ruby example in Listing 10-1, we saw how object data could change while you try to log it on a separate thread. There's no danger of that happening when using atoms to manage state, because each state is immutable. Here's how you could log a zombie's state with `println`:

```
(let [zombie-state @fred]
  (if (>= (:percent-deteriorated zombie-state) 50)
    (future (println (:percent-deteriorated zombie-state)))))
```

The problem with the Ruby example in Listing 10-1 was that it took two steps to read the zombie's two attributes, and some other thread could have changed those attributes in between the two steps. However, by using atoms to refer to immutable data structures, you only have to perform one read, and the data structure returned won't get altered by another thread.

To update the atom so that it refers to a new state, you use `swap!`. This might seem contradictory, because I said that atomic values are unchanging. Indeed, they are! But now we're working with the atom *reference type*, a construct that refers to atomic values. The atomic values don't change, but the reference type can be updated and assigned a new value.

`swap!` receives an atom and a function as arguments. It applies the function to the atom's current state to produce a new value, and then it updates

the atom to refer to this new value. The new value is also returned. Here's how you might increase Fred's cuddle hunger level by one:

```
(swap! fred
       (fn [current-state]
         (merge-with + current-state {:cuddle-hunger-level 1})))
; => {:cuddle-hunger-level 1, :percent-deteriorated 0}
```

Dereferencing `fred` will return the new state:

```
@fred
; => {:cuddle-hunger-level 1, :percent-deteriorated 0}
```

Unlike Ruby, it's not possible for `fred` to be in an inconsistent state, because you can update the hunger level and deterioration percentage at the same time, like this:

```
(swap! fred
       (fn [current-state]
         (merge-with + current-state {:cuddle-hunger-level 1
                                      :percent-deteriorated 1})))
; => {:cuddle-hunger-level 2, :percent-deteriorated 1}
```

This code passes `swap!` a function that takes only one argument, `current-state`. You can also pass `swap!` a function that takes multiple arguments. For example, you could create a function that takes two arguments, a zombie state and the amount by which to increase its cuddle hunger level:

```
(defn increase-cuddle-hunger-level
  [zombie-state increase-by]
  (merge-with + zombie-state {:cuddle-hunger-level increase-by}))
```

Let's test `increase-cuddle-hunger-level` out real quick on a zombie state.

```
(increase-cuddle-hunger-level @fred 10)
; => {:cuddle-hunger-level 12, :percent-deteriorated 1}
```

Note that this code doesn't actually update `fred`, because we're not using `swap!` We're just making a normal function call to `increase-cuddle-hunger-level`, which returns a result.

Now call `swap!` with the additional arguments, and `@fred` will be updated, like this:

```
(swap! fred increase-cuddle-hunger-level 10)
; => {:cuddle-hunger-level 12, :percent-deteriorated 1}

@fred
; => {:cuddle-hunger-level 12, :percent-deteriorated 1}
```

Or you could express the whole thing using Clojure's built-in functions. The `update-in` function takes three arguments: a collection, a vector for identifying which value to update, and a function to update that value. It can also take additional arguments that get passed to the update function. Here are a couple of examples:

```
(update-in {:a {:b 3}} [:a :b] inc)
; => {:a {:b 4}}

(update-in {:a {:b 3}} [:a :b] + 10)
; => {:a {:b 13}}
```

In the first example, you're updating the map `{:a {:b 3}}`. Clojure uses the vector `[:a :b]` to traverse the nested maps; `:a` yields the nested map `{:b 3}`, and `:b` yields the value `3`. Clojure applies the `inc` function to `3` and returns a new map with `3` replaced by `4`. The second example is similar. The only difference is that you're using the addition function and you're supplying `10` as an additional argument; Clojure ends up calling `(+ 3 10)`.

Here's how you can use the `update-in` function to change Fred's state:

```
(swap! fred update-in [:cuddle-hunger-level] + 10)
; => {:cuddle-hunger-level 22, :percent-deteriorated 1}
```

By using atoms, you can retain past state. You can dereference an atom to retrieve State 1, and then update the atom, creating State 2, and still make use of State 1:

```
(let [num (atom 1)
      s1 @num]
  (swap! num inc)
  (println "State 1:" s1)
  (println "Current state:" @num))
; => State 1: 1
; => Current state: 2
```

This code creates an atom named `num`, retrieves its state, updates its state, and then prints its past state and its current state, showing that I wasn't trying to trick you when I said you can retain past state, and therefore you can trust me with all manner of things—including your true name, which I promise to utter only to save you from mortal danger.

This is all interesting and fun, but what happens if two separate threads call `(swap! fred increase-cuddle-hunger-level 1)`? Is it possible for one of the increments to get lost the way it did in the Ruby example at Listing 10-1?

The answer is no! `swap!` implements *compare-and-set* semantics, meaning it does the following internally:

1. It reads the current state of the atom.
2. It then applies the update function to that state.

3. Next, it checks whether the value it read in step 1 is identical to the atom's current value.
4. If it is, then `swap!` updates the atom to refer to the result of step 2.
5. If it isn't, then `swap!` retries, going through the process again with step 1.

This process ensures that no swaps will ever get lost.

One detail to note about `swap!` is that atom updates happen synchronously; they will block their thread. For example, if your update function calls `Thread/sleep 1000` for some reason, the thread will block for at least a second while `swap!` completes.

Sometimes you'll want to update an atom without checking its current value. For example, you might develop a serum that sets a cuddle zombie's hunger level and deterioration back to zero. For those cases, you can use the `reset!` function:

```
(reset! fred {:cuddle-hunger-level 0
              :percent-deteriorated 0})
```

And that covers all the core functionality of atoms! To recap: atoms implement Clojure's concept of state. They allow you to endow a series of immutable values with an identity. They offer a solution to the reference cell and mutual exclusion problems through their compare-and-set semantics. They also allow you to work with past states without fear of them mutating in place.

In addition to these core features, atoms also share two features with the other reference types. You can attach both *watches* and *validators* to atoms. Let's look at those now.

Watches and Validators

Watches allow you to be super creepy and check in on your reference types' every move. Validators allow you to be super controlling and restrict what states are allowable. Both watches and validators are plain ol' functions.

Watches

A *watch* is a function that takes four arguments: a key, the reference being watched, its previous state, and its new state. You can register any number of watches with a reference type.

Let's say that a zombie's shuffle speed (measured in shuffles per hour, or SPH) is dependent on its hunger level and deterioration. Here's how you'd calculate it, multiplying the cuddle hunger level by how whole it is:

```
(defn shuffle-speed
  [zombie]
  (* (:cuddle-hunger-level zombie)
     (- 100 (:percent-deteriorated zombie))))
```

Let's also say that you want to be alerted whenever a zombie's shuffle speed reaches the dangerous level of 5,000 SPH. Otherwise, you want to be told that everything's okay. Here's a watch function you could use to print a warning message if the SPH is above 5,000 and print an all's-well message otherwise:

```
(defn shuffle-alert
  [key watched old-state new-state]
  (let [sph (shuffle-speed new-state)]
    (if (> sph 5000)
      (do
        (println "Run, you fool!")
        (println "The zombie's SPH is now " sph)
        (println "This message brought to your courtesy of " key))
      (do
        (println "All's well with " key)
        (println "Cuddle hunger: " (:cuddle-hunger-level new-state))
        (println "Percent deteriorated: " (:percent-deteriorated new-state))
        (println "SPH: " sph)))))
```

Watch functions take four arguments: a key that you can use for reporting, the atom being watched, the state of the atom before its update, and the state of the atom after its update. This watch function calculates the shuffle speed of the new state and prints a warning message if it's too high and an all's-well message when the shuffle speed is safe, as mentioned above. In both sets of messages, the `key` is used to let you know the source of the message.

You can attach this function to `fred` with `add-watch`. The general form of `add-watch` is (`add-watch` *`ref key watch-fn`*). In this example, we're resetting `fred`'s state, adding the `shuffle-alert` watch function, and then updating `fred`'s state a couple of times to trigger `shuffle-alert`:

```
(reset! fred {:cuddle-hunger-level 22
              :percent-deteriorated 2})
(add-watch fred :fred-shuffle-alert shuffle-alert)
(swap! fred update-in [:percent-deteriorated] + 1)
; => All's well with  :fred-shuffle-alert
; => Cuddle hunger:  22
; => Percent deteriorated:  3
; => SPH:  2134

(swap! fred update-in [:cuddle-hunger-level] + 30)
; => Run, you fool!
; => The zombie's SPH is now 5044
; => This message brought to your courtesy of :fred-shuffle-alert
```

This example watch function didn't use `watched` or `old-state`, but they're there for you if the need arises. Now let's cover validators.

Validators

Validators let you specify what states are allowable for a reference. For example, here's a validator that you could use to ensure that a zombie's `:percent-deteriorated` is between 0 and 100:

```
(defn percent-deteriorated-validator
  [{:keys [percent-deteriorated]}]
  (and (>= percent-deteriorated 0)
       (<= percent-deteriorated 100)))
```

As you can see, the validator takes only one argument. When you add a validator to a reference, the reference is modified so that, whenever it's updated, it will call this validator with the value returned from the update function as its argument. If the validator fails by returning `false` or throwing an exception, the reference won't change to point to the new value.

You can attach a validator during atom creation:

```
(def bobby
  (atom
   {:cuddle-hunger-level 0 :percent-deteriorated 0}
    :validator percent-deteriorated-validator))
(swap! bobby update-in [:percent-deteriorated] + 200)
; This throws "Invalid reference state"
```

In this example, `percent-deteriorated-validator` returned `false` and the atom update failed.

You can throw an exception to get a more descriptive error message:

```
(defn percent-deteriorated-validator
  [{:keys [percent-deteriorated]}]
  (or (and (>= percent-deteriorated 0)
           (<= percent-deteriorated 100))
      (throw (IllegalStateException. "That's not mathy!"))))
(def bobby
  (atom
   {:cuddle-hunger-level 0 :percent-deteriorated 0}
    :validator percent-deteriorated-validator))
(swap! bobby update-in [:percent-deteriorated] + 200)
; This throws "IllegalStateException That's not mathy!"
```

Pretty great! Now let's look at refs.

Refs

Atoms are ideal for managing the state of independent identities. Sometimes, though, we need to express that an event should update the state of more than one identity simultaneously. *Refs* are the perfect tool for this scenario.

A classic example of this is recording sock gnome transactions. As we all know, sock gnomes take a single sock from every clothes dryer around the world. They use these socks to incubate their young. In return for this "gift," sock gnomes protect your home from El Chupacabra. If you haven't been visited by El Chupacabra lately, you have sock gnomes to thank.

To model sock transfers, we need to express that a dryer has lost a sock and a gnome has gained a sock simultaneously. One moment the sock belongs to the dryer; the next it belongs to the gnome. The sock should never appear to belong to both the dryer and the gnome, nor should it appear to belong to neither.

Modeling Sock Transfers

You can model this sock transfer with refs. Refs allow you to update the state of multiple identities using transaction semantics. These transactions have three features:

- They are *atomic*, meaning that all refs are updated or none of them are.
- They are *consistent*, meaning that the refs always appear to have valid states. A sock will always belong to a dryer or a gnome, but never both or neither.
- They are *isolated*, meaning that transactions behave as if they executed serially; if two threads are simultaneously running transactions that alter the same ref, one transaction will retry. This is similar to the compare-and-set semantics of atoms.

You might recognize these as the *A*, *C*, and *I* in the ACID properties of database transactions. You can think of refs as giving you the same concurrency safety as database transactions, only with in-memory data.

Clojure uses *software transactional memory (STM)* to implement this behavior. STM is very cool, but when you're starting with Clojure, you don't need to know much about it; you just need to know how to use it, which is what this section shows you.

Let's start transferring some socks! First, you'll need to code up some sock- and gnome-creation technology. The following code defines some sock varieties, then defines a couple of helper functions: `sock-count` will be

used to help keep track of how many of each kind of sock belongs to either a gnome or a dryer, and `generate-sock-gnome` creates a fresh, sockless gnome:

```
(def sock-varieties
  #{"darned" "argyle" "wool" "horsehair" "mulleted"
    "passive-aggressive" "striped" "polka-dotted"
    "athletic" "business" "power" "invisible" "gollumed"})

(defn sock-count
  [sock-variety count]
  {:variety sock-variety
   :count count})

(defn generate-sock-gnome
  "Create an initial sock gnome state with no socks"
  [name]
  {:name name
   :socks #{}})
```

Now you can create your actual refs. The gnome will have 0 socks. The dryer, on the other hand, will have a set of sock pairs generated from the set of sock varieties. Here are our refs:

```
(def sock-gnome (ref (generate-sock-gnome "Barumpharumph")))
(def dryer (ref {:name "LG 1337"
                 :socks (set (map #(sock-count % 2) sock-varieties))}))
```

You can dereference refs just like you can dereference atoms. In this example, the order of your socks will probably be different because we're using an unordered set:

```
(:socks @dryer)
; => #{{:variety "passive-aggressive", :count 2} {:variety "power", :count 2}
       {:variety "athletic", :count 2} {:variety "business", :count 2}
       {:variety "argyle", :count 2} {:variety "horsehair", :count 2}
       {:variety "gollumed", :count 2} {:variety "darned", :count 2}
       {:variety "polka-dotted", :count 2} {:variety "wool", :count 2}
       {:variety "mulleted", :count 2} {:variety "striped", :count 2}
       {:variety "invisible", :count 2}}
```

Now everything's in place to perform the transfer. We'll want to modify the `sock-gnome` ref to show that it has gained a sock and modify the `dryer` ref to show that it's lost a sock. You modify refs using `alter`, and you must use `alter` within a transaction. `dosync` initiates a transaction and defines its extent; you put all transaction operations in its body. Here we use these tools to define a `steal-sock` function, and then call it on our two refs:

```
(defn steal-sock
  [gnome dryer]
  (dosync
```

```
  (when-let [pair (some #(if (= (:count %) 2) %) (:socks @dryer))]
    (let [updated-count (sock-count (:variety pair) 1)]
      (alter gnome update-in [:socks] conj updated-count)
      (alter dryer update-in [:socks] disj pair)
      (alter dryer update-in [:socks] conj updated-count)))))
(steal-sock sock-gnome dryer)

(:socks @sock-gnome)
; => #{{:variety "passive-aggressive", :count 1}}
```

Now the gnome has one passive-aggressive sock, and the dryer has one less (your gnome may have stolen a different sock because the socks are stored in an unordered set). Let's make sure all passive-aggressive socks are accounted for:

```
(defn similar-socks
  [target-sock sock-set]
  (filter #(= (:variety %) (:variety target-sock)) sock-set))

(similar-socks (first (:socks @sock-gnome)) (:socks @dryer))
; => ({:variety "passive-aggressive", :count 1})
```

There are a couple of details to note here: when you `alter` a ref, the change isn't immediately visible outside of the current transaction. This is what lets you call `alter` on the `dryer` twice within a transaction without worrying about whether `dryer` will be read in an inconsistent state. Similarly, if you `alter` a ref and then `deref` it within the same transaction, the `deref` will return the new state.

Here's an example to demonstrate this idea of in-transaction state:

```
(def counter (ref 0))
(future
  (dosync
   (alter counter inc)
   (println @counter)
   (Thread/sleep 500)
   (alter counter inc)
   (println @counter)))
(Thread/sleep 250)
(println @counter)
```

This prints 1, 0 , and 2, in that order. First, you create a ref, `counter`, which holds the number 0. Then you use `future` to create a new thread to run a transaction on. On the transaction thread, you increment the counter and print it, and the number 1 gets printed. Meanwhile, the main thread waits 250 milliseconds and prints the counter's value, too. However, the value of counter on the main thread is still 0—the main thread is outside of the transaction and doesn't have access to the transaction's state. It's like the transaction has its own private area for trying out changes to the state, and the rest of the world can't know about them until the transaction

is done. This is further illustrated in the transaction code: after it prints the first time, it increments the counter again from 1 to 2 and prints the result, 2.

The transaction will try to commit its changes only when it ends. The commit works similarly to the compare-and-set semantics of atoms. Each ref is checked to see whether it's changed since you first tried to alter it. If *any* of the refs have changed, then *none* of the refs is updated and the transaction is retried. For example, if Transaction A and Transaction B are both attempted at the same time and events occur in the following order, Transaction A will be retried:

1. Transaction A: alter gnome
2. Transaction B: alter gnome
3. Transaction B: alter dryer
4. Transaction B: alter dryer
5. Transaction B: commit—successfully updates gnome and dryer
6. Transaction A: alter dryer
7. Transaction A: alter dryer
8. Transaction A: commit—fails because dryer and gnome have changed; retries.

And there you have it! Safe, easy, concurrent coordination of state changes. But that's not all! Refs have one more trick up their suspiciously long sleeve: `commute`.

commute

`commute` allows you to update a ref's state within a transaction, just like `alter`. However, its behavior at commit time is completely different. Here's how `alter` behaves:

1. Reach outside the transaction and read the ref's current state.
2. Compare the current state to the state the ref started with within the transaction.
3. If the two differ, make the transaction retry.
4. Otherwise, commit the altered ref state.

`commute`, on the other hand, behaves like this at commit time:

1. Reach outside the transaction and read the ref's current state.
2. Run the `commute` function again using the current state.
3. Commit the result.

As you can see, `commute` doesn't ever force a transaction retry. This can help improve performance, but it's important that you only use `commute` when you're sure that it's not possible for your refs to end up in an invalid state. Let's look at examples of safe and unsafe uses of `commute`.

Here's an example of a safe use. The `sleep-print-update` function returns the updated state but also sleeps the specified number of milliseconds so we can force transaction overlap. It prints the state that it's attempting to update so we can gain insight into what's going on:

```
(defn sleep-print-update
  [sleep-time thread-name update-fn]
  (fn [state]
    (Thread/sleep sleep-time)
    (println (str thread-name ": " state))
    (update-fn state)))
(def counter (ref 0))
(future (dosync (commute counter (sleep-print-update 100 "Thread A" inc))))
(future (dosync (commute counter (sleep-print-update 150 "Thread B" inc))))
```

Here's a timeline of what prints:

```
Thread A: 0 | 100ms
Thread B: 0 | 150ms
Thread A: 0 | 200ms
Thread B: 1 | 300ms
```

Notice that the last printed line reads `Thread B: 1`. That means that `sleep-print-update` receives `1` as the argument for state the second time it runs. That makes sense, because Thread A has committed its result by that point. If you dereference `counter` after the transactions run, you'll see that the value is `2`.

Now, here's an example of unsafe commuting:

```
(def receiver-a (ref #{}))
(def receiver-b (ref #{}))
(def giver (ref #{1}))
(do (future (dosync (let [gift (first @giver)]
                      (Thread/sleep 10)
                      (commute receiver-a conj gift)
                      (commute giver disj gift))))
    (future (dosync (let [gift (first @giver)]
                      (Thread/sleep 50)
                      (commute receiver-b conj gift)
                      (commute giver disj gift)))))

@receiver-a
; => #{1}

@receiver-b
; => #{1}

@giver
; => #{}
```

The `1` was given to both `receiver-a` and `receiver-b`, and you've ended up with two instances of `1`, which isn't valid for your program. What's different

about this example is that the functions that are applied, essentially `#(conj % gift)` and `#(disj % gift)`, are derived from the state of `giver`. Once `giver` changes, the derived functions produce an invalid state, but `commute` doesn't care that the resulting state is invalid and commits the result anyway. The lesson here is that although `commute` can help speed up your programs, you have to be judicious about when to use it.

Now you're ready to start using refs safely and sanely. Refs have a few more nuances that I won't cover here, but if you're curious about them, you can research the `ensure` function and the phenomenon *write skew*.

On to the final reference type that this book covers: *vars*.

Vars

You've already learned a bit about vars in Chapter 6. To recap briefly, *vars* are associations between symbols and objects. You create new vars with `def`.

Although vars aren't used to manage state in the same way as atoms and refs, they do have a couple of concurrency tricks: you can dynamically bind them, and you can alter their roots. Let's look at dynamic binding first.

Dynamic Binding

When I first introduced `def`, I implored you to treat it as if it's defining a constant. It turns out that vars are a bit more flexible than that: you can create a *dynamic* var whose binding can be changed. Dynamic vars can be useful for creating a global name that should refer to different values in different contexts.

Creating and Binding Dynamic Vars

First, create a dynamic var:

```
(def ^:dynamic *notification-address* "dobby@elf.org")
```

Notice two important details here. First, you use `^:dynamic` to signal to Clojure that a var is dynamic. Second, the var's name is enclosed by asterisks. Lispers call these *earmuffs*, which is adorable. Clojure requires you to enclose the names of dynamic vars in earmuffs. This helps signal the var's *dynamicaltude* to other programmers.

Unlike regular vars, you can temporarily change the value of dynamic vars by using `binding`:

```
(binding [*notification-address* "test@elf.org"]
  *notification-address*)
; => "test@elf.org"
```

You can also stack bindings (just like you can with `let`):

```
(binding [*notification-address* "tester-1@elf.org"]
  (println *notification-address*)
```

```
  (binding [*notification-address* "tester-2@elf.org"]
    (println *notification-address*))
  (println *notification-address*))
; => tester-1@elf.org
; => tester-2@elf.org
; => tester-1@elf.org
```

Now that you know how to dynamically bind a var, let's look at a real-world application.

Dynamic Var Uses

Let's say you have a function that sends a notification email. In this example, we'll just return a string but pretend that the function actually sends the email:

```
(defn notify
  [message]
  (str "TO: " *notification-address* "\n"
       "MESSAGE: " message))
(notify "I fell.")
; => "TO: dobby@elf.org\nMESSAGE: I fell."
```

What if you want to test this function without spamming Dobby every time your specs run? Here comes `binding` to the rescue:

```
(binding [*notification-address* "test@elf.org"]
  (notify "test!"))
; => "TO: test@elf.org\nMESSAGE: test!"
```

Of course, you could have just defined `notify` to take an email address as an argument. In fact, that's often the right choice. Why would you want to use dynamic vars instead?

Dynamic vars are most often used to name a resource that one or more functions target. In this example, you can view the email address as a resource that you write to. In fact, Clojure comes with a ton of built-in dynamic vars for this purpose. `*out*`, for example, represents the standard output for print operations. In your program, you could re-bind `*out*` so that print statements write to a file, like so:

```
(binding [*out* (clojure.java.io/writer "print-output")]
  (println "A man who carries a cat by the tail learns
something he can learn in no other way.
-- Mark Twain"))
(slurp "print-output")
; => A man who carries a cat by the tail learns
     something he can learn in no other way.
     -- Mark Twain
```

This is much less burdensome than passing an output destination to every invocation of `println`. Dynamic vars are a great way to specify a common resource while retaining the flexibility to change it on an ad hoc basis.

Dynamic vars are also used for configuration. For example, the built-in var `*print-length*` allows you to specify how many items in a collection Clojure should print:

```
(println ["Print" "all" "the" "things!"])
; => [Print all the things!]

(binding [*print-length* 1]
  (println ["Print" "just" "one!"]))
; => [Print ...]
```

Finally, it's possible to `set!` dynamic vars that have been bound. Whereas the examples you've seen so far allow you to convey information *in* to a function without having to pass in the information as an argument, `set!` allows you convey information *out* of a function without having to return it as an argument.

For example, let's say you're a telepath, but your mind-reading powers are a bit delayed. You can read people's thoughts only after the moment when it would have been useful for you to know them. Don't feel too bad, though; you're still a telepath, which is awesome. Anyway, say you're trying to cross a bridge guarded by a troll who will eat you if you don't answer his riddle. His riddle is "What number between 1 and 2 am I thinking of?" In the event that the troll devours you, you can at least die knowing what the troll was actually thinking.

In this example, you create the dynamic var `*troll-thought*` to convey the troll's thought out of the `troll-riddle` function:

```
(def ^:dynamic *troll-thought* nil)
(defn troll-riddle
  [your-answer]
  (let [number "man meat"]
❶   (when (thread-bound? #'*troll-thought*)
❷     (set! *troll-thought* number))
    (if (= number your-answer)
      "TROLL: You can cross the bridge!"
      "TROLL: Time to eat you, succulent human!")))
```

```
(binding [*troll-thought* nil]
  (println (troll-riddle 2))
  (println "SUCCULENT HUMAN: Oooooh! The answer was" *troll-thought*))

; => TROLL: Time to eat you, succulent human!
; => SUCCULENT HUMAN: Oooooh! The answer was man meat
```

You use the `thread-bound?` function at ❶ to check that the var has been bound, and if it has, you `set! *troll-thought*` to the troll's thought at ❷.

The var returns to its original value outside of binding:

```
*troll-thought*
; => nil
```

Notice that you have to pass `#'*troll-thought*` (including `#'`), not `*troll-thought*`, to the function `thread-bound?`. This is because `thread-bound?` takes the var itself as an argument, not the value it refers to.

Per-Thread Binding

One final point to note about binding: if you access a dynamically bound var from within a manually created thread, the var will evaluate to the original value. If you're new to Clojure (and Java), this feature won't be immediately relevant; you can probably skip this section and come back to it later.

Ironically, this binding behavior prevents us from easily creating a fun demonstration in the REPL, because the REPL binds `*out*`. It's as if all the code you run in the REPL is implicitly wrapped in something like `(binding [*out* repl-printer] your-code`. If you create a new thread, `*out*` won't be bound to the REPL printer.

The following example uses some basic Java interop. Even if it looks unfamiliar, the gist of the following code should be clear, and you'll learn exactly what's going on in Chapter 12.

This code prints output to the REPL:

```
(.write *out* "prints to repl")
; => prints to repl
```

The following code doesn't print output to the REPL, because `*out*` is not bound to the REPL printer:

```
(.start (Thread. #(.write *out* "prints to standard out")))
```

You can work around this by using this goofy code:

```
(let [out *out*]
  (.start
   (Thread. #(binding [*out* out]
               (.write *out* "prints to repl from thread")))))
```

Or you can use `bound-fn`, which carries all the current bindings to the new thread:

```
(.start (Thread. (bound-fn [] (.write *out* "prints to repl from thread"))))
```

The `let` binding captures `*out*` so we can then rebind it in the child thread, which is goofy as hell. The point is that bindings don't get passed on to *manually* created threads. They do, however, get passed on to futures. This is called *binding conveyance*. Throughout this chapter, we've been printing from futures without any problem, for example.

That's it for dynamic binding. Let's turn our attention to the last var topic: altering var *roots*.

Altering the Var Root

When you create a new var, the initial value that you supply is its *root*:

```
(def power-source "hair")
```

In this example, `"hair"` is the root value of `power-source`. Clojure lets you permanently change this root value with the function `alter-var-root`:

```
(alter-var-root #'power-source (fn [_] "7-eleven parking lot"))
power-source
; => "7-eleven parking lot"
```

Just like when using `swap!` to update an atom or `alter!` to update a ref, you use `alter-var-root` along with a function to update the state of a var. In this case, the function is just returning a new string that bears no relation to the previous value, unlike the `alter!` examples where we used `inc` to derive a new number from the current number.

You'll hardly ever want to do this. You especially don't want to do this to perform simple variable assignment. If you did, you'd be going out of your way to create the binding as a mutable variable, which goes against Clojure's philosophy; it's best to use the functional programming techniques you learned in Chapter 5.

You can also temporarily alter a var's root with `with-redefs`. This works similarly to binding except the alteration will appear in child threads. Here's an example:

```
(with-redefs [*out* *out*]
        (doto (Thread. #(println "with redefs allows me to show up in the REPL"))
          .start
          .join))
```

Using `with-redefs` may be more appropriate than using `bindings` for setting up a test environment. It's also more widely applicable, in that you can use it for any var, not just dynamic ones.

Now you know all about vars! Try not to hurt yourself or anyone you know with them.

Stateless Concurrency and Parallelism with pmap

So far, this chapter has focused on tools that are designed to mitigate the risks inherent in concurrent programming. You've learned about the dangers born of shared access to mutable state and how Clojure implements a reconceptualization of state that helps you write concurrent programs safely.

Often, though, you'll want to concurrent-ify tasks that are completely independent of each other. There is no shared access to a mutable state; therefore, there are no risks to running the tasks concurrently and you don't have to bother with using any of the tools I've just been blabbing on about.

As it turns out, Clojure makes it easy for you to write code for achieving stateless concurrency. In this section, you'll learn about `pmap`, which gives you concurrency performance benefits virtually for free.

`map` is a perfect candidate for parallelization: when you use it, all you're doing is deriving a new collection from an existing collection by applying a function to each element of the existing collection. There's no need to maintain state; each function application is completely independent. Clojure makes it easy to perform a parallel map with `pmap`. With `pmap`, Clojure handles the running of each application of the mapping function on a separate thread.

To compare `map` and `pmap`, we need a lot of example data, and to generate this data, we'll use the `repeatedly` function. This function takes another function as an argument and returns a lazy sequence. The elements of the lazy sequence are generated by calling the passed function, like this:

```
(defn always-1
  []
  1)
(take 5 (repeatedly always-1))
; => (1 1 1 1 1)
```

Here's how you'd create a lazy seq of random numbers between 0 and 9:

```
(take 5 (repeatedly (partial rand-int 10)))
; => (1 5 0 3 4)
```

Let's use `repeatedly` to create example data that consists of a sequence of 3,000 random strings, each 7,000 characters long. We'll compare `map` and `pmap` by using them to run `clojure.string/lowercase` on the `orc-names` sequence created here:

```
(def alphabet-length 26)

;; Vector of chars, A-Z
(def letters (mapv (comp str char (partial + 65)) (range alphabet-length)))
```

```
(defn random-string
  "Returns a random string of specified length"
  [length]
  (apply str (take length (repeatedly #(rand-nth letters)))))

(defn random-string-list
  [list-length string-length]
  (doall (take list-length (repeatedly (partial random-string string-length)))))

(def orc-names (random-string-list 3000 7000))
```

Because `map` and `pmap` are lazy, we have to force them to be realized. We don't want the result to be printed to the REPL, though, because that would take forever. The `dorun` function does just what we need: it realizes the sequence but returns `nil`:

```
(time (dorun (map clojure.string/lower-case orc-names)))
; => "Elapsed time: 270.182 msecs"

(time (dorun (pmap clojure.string/lower-case orc-names)))
; => "Elapsed time: 147.562 msecs"
```

The serial execution with `map` took about 1.8 times longer than `pmap`, and all you had to do was add one extra letter! Your performance may be even better, depending on the number of cores your computer has; this code was run on a dual-core machine.

You might be wondering why the parallel version didn't take exactly half as long as the serial version. After all, it should take two cores only half as much time as a single core, shouldn't it? The reason is that there's always some overhead involved with creating and coordinating threads. Sometimes, in fact, the time taken by this overhead can dwarf the time of each function application, and `pmap` can actually take longer than `map`. Figure 10-3 shows how you can visualize this.

Figure 10-3: Parallelization overhead can dwarf task time, resulting in a performance decrease.

We can see this effect at work if we run a function on 20,000 abbreviated orc names, each 300 characters long:

```
(def orc-name-abbrevs (random-string-list 20000 300))
(time (dorun (map clojure.string/lower-case orc-name-abbrevs)))
; => "Elapsed time: 78.23 msecs"
(time (dorun (pmap clojure.string/lower-case orc-name-abbrevs)))
; => "Elapsed time: 124.727 msecs"
```

Now `pmap` actually takes 1.6 times *longer*.

The solution to this problem is to increase the *grain size*, or the amount of work done by each parallelized task. In this case, the task is to apply the mapping function to one element of the collection. Grain size isn't measured in any standard unit, but you'd say that the grain size of `pmap` is one by default. Increasing the grain size to two would mean that you're applying the mapping function to two elements instead of one, so the thread that the task is on is doing more work. Figure 10-4 shows how an increased grain size can improve performance.

Figure 10-4: Visualizing grain size in relation to parallelization overhead

To actually accomplish this in Clojure, you can increase the grain size by making each thread apply `clojure.string/lower-case` to multiple elements instead of just one, using `partition-all`. `partition-all` takes a seq and divides it into seqs of the specified length:

```
(def numbers [1 2 3 4 5 6 7 8 9 10])
(partition-all 3 numbers)
; => ((1 2 3) (4 5 6) (7 8 9) (10))
```

Now suppose you started out with code that looked like this:

```
(pmap inc numbers)
```

In this case, the grain size is one because each thread applies `inc` to an element.

Now suppose you changed the code to this:

```
(pmap (fn [number-group] (doall (map inc number-group)))
      (partition-all 3 numbers))
; => ((2 3 4) (5 6 7) (8 9 10) (11))
```

There are a few things going on here. First, you've now increased the grain size to three because each thread now executes three applications of the `inc` function instead of one. Second, notice that you have to call `doall` within the mapping function. This forces the lazy sequence returned by `(map inc number-group)` to be realized within the thread. Third, we need to ungroup the result. Here's how we can do that:

```
(apply concat
       (pmap (fn [number-group] (doall (map inc number-group)))
             (partition-all 3 numbers)))
```

Using this technique, we can increase the grain size of the orc name lowercase-ification so each thread runs `clojure.string/lower-case` on 1,000 names instead of just one:

```
(time
 (dorun
  (apply concat
         (pmap (fn [name] (doall (map clojure.string/lower-case name)))
               (partition-all 1000 orc-name-abbrevs)))))
; => "Elapsed time: 44.677 msecs"
```

Once again the parallel version takes nearly half the time. Just for fun, we can generalize this technique into a function called `ppmap`, for *partitioned pmap*. It can receive more than one collection, just like `map`:

```
(defn ppmap
  "Partitioned pmap, for grouping map ops together to make parallel
  overhead worthwhile"
  [grain-size f & colls]
  (apply concat
   (apply pmap
          (fn [& pgroups] (doall (apply map f pgroups)))
          (map (partial partition-all grain-size) colls))))
(time (dorun (ppmap 1000 clojure.string/lower-case orc-name-abbrevs)))
; => "Elapsed time: 44.902 msecs"
```

I don't know about you, but I think this stuff is just fun. For even more fun, check out the clojure.core.reducers library (*http://clojure.org/reducers/*). This library provides alternative implementations of seq functions like `map` and `reduce` that are usually speedier than their cousins in `clojure.core`. The

trade-off is that they're not lazy. Overall, the clojure.core.reducers library offers a more refined and composable way of creating and using functions like `ppmap`.

Summary

In this chapter, you learned more than most people know about safely handling concurrent tasks. You learned about the metaphysics that underlies Clojure's reference types. In Clojure metaphysics, state is the value of an identity at a point in time, and identity is a handy way to refer to a succession of values produced by some process. Values are atomic in the same way numbers are atomic. They're immutable, and this makes them safe to work with concurrently; you don't have to worry about other threads changing them while you're using them.

The atom reference type allows you to create an identity that you can safely update to refer to new values using `swap!` and `reset!`. The ref reference type is handy when you want to update more than one identity using transaction semantics, and you update it with `alter!` and `commute!`.

Additionally, you learned how to increase performance by performing stateless data transformations with `pmap` and the core.reducers library. Woohoo!

Exercises

1. Create an atom with the initial value 0, use `swap!` to increment it a couple of times, and then dereference it.
2. Create a function that uses futures to parallelize the task of downloading random quotes from *http://www.braveclojure.com/random-quote* using `(slurp "http://www.braveclojure.com/random-quote")`. The futures should update an atom that refers to a total word count for all quotes. The function will take the number of quotes to download as an argument and return the atom's final value. Keep in mind that you'll need to ensure that all futures have finished before returning the atom's final value. Here's how you would call it and an example result:

```
(quote-word-count 5)
; => {"ochre" 8, "smoothie" 2}
```

3. Create representations of two characters in a game. The first character has 15 hit points out of a total of 40. The second character has a healing potion in his inventory. Use refs and transactions to model the consumption of the healing potion and the first character healing.

11

MASTERING CONCURRENT PROCESSES WITH CORE.ASYNC

One day, while you are walking down the street, you will be surprised, intrigued, and a little disgusted to discover a hot dog vending machine. Your scalp tingling with guilty curiosity, you won't be able to help yourself from pulling out three dollars and seeing if this contraption actually works. After accepting your money with a click and a whir, it pops out a fresh hot dog, bun and all.

The vending machine exhibits simple behavior: when it receives money, it releases a hot dog and then gets ready for the next purchase. When it's out of hot dogs, it stops. All around us are hot dog vending machines in different guises—independent entities concurrently responding to events in the world. The espresso machine at your favorite coffee shop, the pet hamster you loved as a child—everything can be deconstructed into a set of behaviors that follow the general form "when *x* happens, do *y*." Even the programs we write are just glorified hot dog vending machines, each one an

independent process waiting for the next event, whether it's a keystroke, a timeout, or the arrival of data on a socket.

Clojure's core.async library allows you to create multiple independent processes within a single program. This chapter describes a useful model for thinking about this style of programming as well as the practical details you need to know to actually write code. You'll learn how to use channels to communicate between independent processes created by go blocks and `thread`; a bit about how Clojure manages threads efficiently with parking and blocking; how to use `alts!!`; and a more straightforward way of creating queues. Finally, you'll learn how to kick callbacks in the butt with process pipelines.

Getting Started with Processes

At the heart of core.async is the *process*, a concurrently running unit of logic that responds to events. The process corresponds to our mental model of the real world: entities interact with and respond to each other independently without some kind of central control mechanism pulling the strings. You put your money in the machine, and out comes a hot dog, all without the Illuminati or Big Brother orchestrating the whole thing. This differs from the view of concurrency you've been exploring so far, where you've defined tasks that are either mere extensions of the main thread of control (for example, achieving data parallelism with `pmap`) or tasks that you have no interest in communicating with (like one-off tasks created with `future`).

It might be strange to think of a vending machine as a process: vending machines are noun-y and thing-y, and processes are verb-y and do-y. To get in the right mindset, try defining real-world objects as the sum of their event-driven behavior. When a seed gets watered, it sprouts; when a mother looks at her newborn child, she feels love; and when you watch *Star Wars Episode I*, you are filled with anger and despair. If you want to get super philosophical, consider whether it's possible to define every thing's essence as the set of the events it recognizes and how it responds. Is reality just the composition of hot dog vending machines?

Anyway, enough of my yakking! Let's move from the theoretical to the concrete by creating some simple processes. First, create a new Leiningen project called *playsync* with `lein new app playsync`. Then, open the file *project.clj* and add core.async to the `:dependencies` vector so it reads as follows:

```
[[org.clojure/clojure "1.7.0"]
[org.clojure/core.async "0.1.346.0-17112a-alpha"]]
```

NOTE *It's possible that the core.async version has advanced since I wrote this. For the latest version, check the core.async GitHub project page. But for the purpose of these exercises, please use the version listed here.*

Next, open *src/playsync/core.clj* and make it look like this:

```
(ns playsync.core
  (:require [clojure.core.async
             :as a
             :refer [>! <! >!! <!! go chan buffer close! thread
                     alts! alts!! timeout]]))
```

Now when you open this in a REPL, you'll have the most frequently used core.async functions at your disposal. Great! Before creating something as sophisticated and revolutionary as a hot dog vending machine, create a process that simply prints the message it receives:

```
(def echo-chan (chan))
(go (println (<! echo-chan)))
(>!! echo-chan "ketchup")
; => true
; => ketchup
```

At the first line of code, you used the `chan` function to create a *channel* named `echo-chan`. Channels communicate *messages*. You can *put* messages on a channel and *take* messages off a channel. Processes *wait* for the completion of put and take—these are the events that processes respond to. You can think of processes as having two rules: 1) when trying to put a message on a channel or take a message off of it, wait and do nothing until the put or take succeeds, and 2) when the put or take succeeds, continue executing.

On the next line, you used `go` to create a new process. Everything within the `go` expression—called a *go block*—runs concurrently on a separate thread. Go blocks run your processes on a thread pool that contains a number of threads equal to two plus the number of cores on your machine, which means your program doesn't have to create a new thread for each process. This often results in better performance because you avoid the overhead associated with creating threads.

In this case, the process (`println (<! echo-chan)`) expresses "when I take a message from `echo-chan`, print it." The process is shunted to another thread, freeing up the current thread and allowing you to continue interacting with the REPL.

In the expression (`<! echo-chan`), `<!` is the *take* function. It listens to the channel you give it as an argument, and the process it belongs to waits until another process puts a message on the channel. When `<!` retrieves a value, the value is returned and the `println` expression is executed.

The expression (`>!! echo-chan "ketchup"`) *puts* the string `"ketchup"` on `echo-chan` and returns `true`. When you put a message on a channel, the process blocks until another process takes the message. In this case, the REPL process didn't have to wait at all, because there was already a process listening to the channel, waiting to take something off it. However, if you do the following, your REPL will block indefinitely:

```
(>!! (chan) "mustard")
```

You've created a new channel and put something on it, but there's no process listening to that channel. Processes don't just wait to receive messages; they also wait for the messages they put on a channel to be taken.

Buffering

It's worth noting that the previous exercise contained *two* processes: the one you created with `go` and the REPL process. These processes don't have explicit knowledge of each other, and they act independently.

Let's imagine that these processes take place in a diner. The REPL is the ketchup chef, and when he's done with a batch, he belts out, "Ketchup!" It's entirely possible that the rest of the staff is outside admiring the latest batch of oregano in their organic garden, and the chef just sits and waits until someone shows up to take his ketchup. On the flip side, the `go` process represents one of the staff, and he's waiting patiently for something to respond to. It could be that nothing ever happens, and he just waits indefinitely until the restaurant closes.

This situation seems a little silly: what self-respecting ketchup chef would just sit and wait for someone to take his latest batch before making more ketchup? To avoid this tragedy, you can create buffered channels:

```
(def echo-buffer (chan 2))
(>!! echo-buffer "ketchup")
; => true
(>!! echo-buffer "ketchup")
; => true
(>!! echo-buffer "ketchup")
; This blocks because the channel buffer is full
```

(Be careful evaluating the last (`>!! echo-buffer "ketchup"`) because it will block your REPL. If you're using a Leiningen REPL, CTRL-C will unblock it.)

In this case, you've created a channel with buffer size 2. That means you can put two values on the channel without waiting, but putting a third one on means the process will wait until another process takes a value from the channel. You can also create *sliding* buffers with `sliding-buffer`,

which drops values in a first-in, first-out fashion; and *dropping* buffers with `dropping-buffer`, which discards values in a last-in, first-out fashion. Neither of these buffers will ever cause `>!!` to block.

By using buffers, the master ketchup chef can keep whipping up batches of mouthwatering ketchup without having to wait for his staff to take them away. If he's using a regular buffer, it's like he has a shelf to put all his ketchup batches on; once the shelf is full, he'll still have to wait for space to open up. If he's using a sliding buffer, he'd throw away the oldest batch of ketchup when the shelf is full, slide all the ketchup down, and put the new batch in the vacant space. With a dropping buffer, he'd just knock the freshest batch off of the shelf and put his new batch in that space.

Buffers are just elaborations of the core model: processes are independent, concurrently executing units of logic that respond to events. You can create processes with go blocks and communicate events over channels.

Blocking and Parking

You may have noticed that the take function `<!` used only one exclamation point, whereas the put function `>!!` used two. In fact, both put and take have one-exclamation-point and two-exclamation-point varieties. When do you use which? The simple answer is that you can use one exclamation point inside go blocks, but you have to use two exclamation points outside of them:

	Inside go block	Outside go block
put	`>!` or `>!!`	`>!!`
take	`<!` or `<!!`	`<!!`

It all comes down to efficiency. Because go blocks use a thread pool with a fixed size, you can create 1,000 go processes but use only a handful of threads:

```
(def hi-chan (chan))
(doseq [n (range 1000)]
  (go (>! hi-chan (str "hi " n))))
```

To understand how Clojure accomplishes this, we need to explore how processes *wait*. Waiting is a key aspect of working with core.async processes: we've already established that *put* waits until another process does a *take* on the same channel, and vice versa. In this example, 1,000 processes are waiting for another process to take from `hi-chan`.

There are two varieties of waiting: *parking* and *blocking*. Blocking is the kind of waiting you're familiar with: a thread stops execution until a task is complete. Usually this happens when you're doing some kind of I/O operation. The thread remains alive but doesn't do any work, so you have to create a new thread if you want your program to continue working. In Chapter 9, you learned how to do this with `future`.

Parking frees up the thread so it can keep doing work. Let's say you have one thread and two processes, Process A and Process B. Process A is running on the thread and then waits for a put or take. Clojure moves Process A off the thread and moves Process B onto the thread. If Process B starts waiting and Process A's put or take has finished, then Clojure will move Process B off the thread and put Process A back on it. Parking allows the instructions from multiple processes to interleave on a single thread, similar to the way that using multiple threads allows interleaving on a single core. The implementation of parking isn't important; suffice it to say that it's only possible within go blocks, and it's only possible when you use `>!` and `<!`, or *parking put* and *parking take*. `>!!` and `<!!` are *blocking put* and *blocking take*.

thread

There are definitely times when you'll want to use blocking instead of parking, like when your process will take a long time before putting or taking, and for those occasions you should use `thread`:

```
(thread (println (<!! echo-chan)))
(>!! echo-chan "mustard")
; => true
; => mustard
```

`thread` acts almost exactly like `future`: it creates a new thread and executes a process on that thread. Unlike `future`, instead of returning an object that you can dereference, `thread` returns a channel. When `thread`'s process stops, the process's return value is put on the channel that `thread` returns:

```
(let [t (thread "chili")]
  (<!! t))
; => "chili"
```

In this case, the process doesn't wait for any events; instead, it stops immediately. Its return value is `"chili"`, which gets put on the channel that's bound to `t`. We take from `t`, returning `"chili"`.

The reason you should use `thread` instead of a go block when you're performing a long-running task is so you don't clog your thread pool. Imagine you're running four processes that download humongous files, save them, and then put the file paths on a channel. While the processes are downloading files and saving these files, Clojure can't park their threads. It can park the thread only at the last step, when the process puts the files' paths on a channel. Therefore, if your thread pool has only four threads, all four threads will be used for downloading, and no other process will be allowed to run until one of the downloads finishes.

`go`, `thread`, `chan`, `<!`, `<!!`, `>!`, and `>!!` are the core tools you'll use for creating and communicating with processes. Both put and take will cause a process to wait until its complement is performed on the given channel. `go` allows you to use the parking variants of put and take, which could improve performance.

You should use the blocking variants, along with `thread`, if you're performing long-running tasks before the put or take.

And that should give you everything you need to fulfill your heart's desire and create a machine that turns money into hot dogs.

The Hot Dog Machine Process You've Been Longing For

Behold, your dreams made real!

```
(defn hot-dog-machine
  []
  (let [in (chan)
        out (chan)]
    (go (<! in)
        (>! out "hot dog"))
    [in out]))
```

This function creates an `in` channel for receiving money and an `out` channel for dispensing a hot dog. It then creates an asynchronous process with `go`, which waits for money and then dispenses a hot dog. Finally, it returns the `in` and `out` channels as a vector.

Time for a hot dog!

```
(let [[in out] (hot-dog-machine)]
  (>!! in "pocket lint")
  (<!! out))
; => "hot dog"
```

In this snippet, you use destructuring (covered in Chapter 3) with `let` to bind the `in` and `out` channels to the `in` and `out` symbols. You then put `"pocket lint"` on the `in` channel. The hot dog machine process waits for something, anything, to arrive on the `in` channel; once `"pocket lint"` arrives, the hot dog machine process resumes execution, putting `"hot dog"` on the `out` channel.

Wait a minute . . . that's not right. I mean, yay, free hot dogs, but someone's bound to get upset that the machine's accepting pocket lint as payment. Not only that, but this machine will only dispense one hot dog before shutting down. Let's alter the hot dog machine function so that you can specify how many hot dogs it has and so it only dispenses a hot dog when you give it the number 3:

```
(defn hot-dog-machine-v2
  [hot-dog-count]
  (let [in (chan)
        out (chan)]
    (go (loop [hc hot-dog-count]
          (if (> hc 0)
            (let [input (<! in)]
             ❶(if (= 3 input)
                (do (>! out "hot dog")
```

```
                (recur (dec hc)))
            (do (>! out "wilted lettuce")
                (recur hc))))
        ❷(do (close! in)
            (close! out)))))
    [in out]))
```

There's a lot more code here, but the strategy is straightforward. The new function `hot-dog-machine-v2` allows you to specify the `hot-dog-count`. Within the go block at ❶, it dispenses a hot dog only if the number 3 (meaning three dollars) is placed on the `in` channel; otherwise, it dispenses wilted lettuce, which is definitely not a hot dog. Once a process has taken the output, the hot dog machine process loops back with an updated hot dog count and is ready to receive money again.

When the machine process runs out of hot dogs, the process *closes* the channels at ❷. When you close a channel, you can no longer perform puts on it, and once you've taken all values off a closed channel, any subsequent takes will return `nil`.

Let's give the upgraded hot dog machine a go in Listing 11-1 by putting in money and pocket lint:

```
(let [[in out] (hot-dog-machine-v2 2)]
  (>!! in "pocket lint")
  (println (<!! out))

  (>!! in 3)
  (println (<!! out))

  (>!! in 3)
  (println (<!! out))

  (>!! in 3)
  (<!! out))
; => wilted lettuce
; => hotdog
; => hotdog
; => nil
```

Listing 11-1: Interacting with a robust hot dog vending machine process

First, we try the ol' pocket lint trick and get wilted lettuce. Next, we put in 3 dollars twice and get a hot dog both times. Then, we try to put in another 3 dollars, but that's ignored because the channel is closed; the number 3 is not put on the channel. When we try to take from the `out` channel, we get `nil`, again because the channel is closed. You might notice a couple of interesting details about `hot-dog-machine-v2`. First, it does a put and a take within the same go block. This isn't that unusual, and it's one way you can create a *pipeline* of processes: just make the *in* channel of one process the *out* channel of another. The following example does just that,

passing a string through a series of processes that perform transformations until the string finally gets printed by the last process:

```
(let [c1 (chan)
      c2 (chan)
      c3 (chan)]
  (go (>! c2 (clojure.string/upper-case (<! c1))))
  (go (>! c3 (clojure.string/reverse (<! c2))))
  (go (println (<! c3)))
  (>!! c1 "redrum"))
; => MURDER
```

I'll have more to say about process pipelines and how you can use them instead of callbacks toward the end of the chapter.

Back to Listing 11-1! Another thing to note is that the hot dog machine doesn't accept more money until you've dealt with whatever it's dispensed. This allows you to model state-machine-like behavior, where the completion of channel operations triggers state transitions. For example, you can think of the vending machine as having two states: *ready to receive money* and *dispensed item*. Inserting money and taking the item trigger transitions between the two.

alts!!

The core.async function `alts!!` lets you use the result of the first successful channel operation among a collection of operations. We did something similar to this with delays and futures in "Delays" on page 198. In that example, we uploaded a set of headshots to a headshot-sharing site and notified the headshot owner when the first photo was uploaded. Here's how you'd do the same with `alts!!`:

```
(defn upload
  [headshot c]
  (go (Thread/sleep (rand 100))
      (>! c headshot)))

❶ (let [c1 (chan)
        c2 (chan)
        c3 (chan)]
    (upload "serious.jpg" c1)
    (upload "fun.jpg" c2)
    (upload "sassy.jpg" c3)
❷   (let [[headshot channel] (alts!! [c1 c2 c3])]
      (println "Sending headshot notification for" headshot)))
; => Sending headshot notification for sassy.jpg
```

Here, the `upload` function takes a headshot and a channel, and creates a new process that sleeps for a random amount of time (to simulate the

upload) and then puts the headshot on the channel. The `let` bindings and `upload` function calls beginning at ❶ should make sense: we create three channels and then use them to perform the uploads.

Things get interesting at ❷. The `alts!!` function takes a vector of channels as its argument. This is like saying, "Try to do a blocking take on each of these channels simultaneously. As soon as a take succeeds, return a vector whose first element is the value taken and whose second element is the winning channel." In this case, the channel associated with *sassy.jpg* received a value first. The other channels are still available if you want to take their values and do something with them. All `alts!!` does is take a value from the first channel to have a value; it doesn't touch the other channels.

One cool aspect of `alts!!` is that you can give it a *timeout channel*, which waits the specified number of milliseconds and then closes. It's an elegant mechanism for putting a time limit on concurrent operations. Here's how you could use it with the upload service:

```
(let [c1 (chan)]
  (upload "serious.jpg" c1)
  (let [[headshot channel] (alts!! [c1 (timeout 20)])]
    (if headshot
      (println "Sending headshot notification for" headshot)
      (println "Timed out!"))))
; => Timed out!
```

In this case, we set the timeout to 20 milliseconds. Because the upload didn't finish in that time frame, we got a timeout message.

You can also use `alts!!` to specify put operations. To do that, place a vector inside the vector you pass to `alts!!`, like at ❶ in this example:

```
(let [c1 (chan)
      c2 (chan)]
  (go (<! c2))
❶ (let [[value channel] (alts!! [c1 [c2 "put!"]])]
    (println value)
    (= channel c2)))
; => true
; => true
```

Here you're creating two channels and then creating a process that's waiting to perform a take on `c2`. The vector that you supply to `alts!!` tells it, "Try to do a take on `c1` and try to put `"put!"` on `c2`. If the take on `c1` finishes first, return its value and channel. If the put on `c2` finishes first, return `true` if the put was successful and `false` otherwise." Finally, the result of `value` (which is `true`, because the `c2` channel was open) prints and shows that the channel returned was indeed `c2`.

Like `<!!` and `>!!`, `alts!!` has a parking alternative, `alts!`, which you can use inside go blocks. `alts!` is a nice way to exercise some choice over which of a group of channels you put or take from. It still performs puts and takes, so the same reasons to use the parking or blocking variation apply.

And that covers the core.async basics! The rest of the chapter explains two common patterns for coordinating processes.

Queues

In "Rolling Your Own Queue" on page 202, you wrote a macro that let you queue futures. Processes let you use a similar technique in a more straightforward manner. Let's say you want to get a bunch of random quotes from a website and write them to a single file. You want to make sure that only one quote is written to a file at a time so the text doesn't get interleaved, so you put your quotes on a queue. Here's the full code:

```
(defn append-to-file
  "Write a string to the end of a file"
  [filename s]
  (spit filename s :append true))

(defn format-quote
  "Delineate the beginning and end of a quote because it's convenient"
  [quote]
  (str "=== BEGIN QUOTE ===\n" quote "=== END QUOTE ===\n\n"))

(defn random-quote
  "Retrieve a random quote and format it"
  []
  (format-quote (slurp "http://www.braveclojure.com/random-quote")))

(defn snag-quotes
  [filename num-quotes]
  (let [c (chan)]
    (go (while true (append-to-file filename (<! c))))
    (dotimes [n num-quotes] (go (>! c (random-quote))))))
```

The functions `append-to-file`, `format-quote`, and `random-quote` have docstrings that explain what they do. `snag-quotes` is where the interesting work happens. First, it creates a channel that's shared between the quote-producing processes and the quote-consuming process. Then it creates a process that uses `while true` to create an infinite loop. On every iteration of the loop, it waits for a quote to arrive on `c` and then appends it to a file. Finally, `snag-quotes` creates a `num-quotes` number of processes that fetch a quote and then put it on `c`. If you evaluate `(snag-quotes "quotes" 2)` and check the *quotes* file in the directory where you started your REPL, it should have two quotes:

```
=== BEGIN QUOTE ===
Nobody's gonna believe that computers are intelligent until they start
coming in late and lying about it.
=== END QUOTE ===
```

```
=== BEGIN QUOTE ===
Give your child mental blocks for Christmas.
=== END QUOTE ===
```

This kind of queuing differs from the example in Chapter 9. In that example, each task was handled in the order it was created. Here, each quote-retrieving task is handled in the order that it finishes. In both cases, you ensure that only one quote at a time is written to a file.

Escape Callback Hell with Process Pipelines

In languages without channels, you need to express the idea "when *x* happens, do *y*" with `callbacks`. In a language like JavaScript, callbacks are a way to define code that executes asynchronously once other code finishes. If you've worked with JavaScript, you've probably spent some time wallowing in *callback hell*.

The reason it's called callback hell is that it's very easy to create dependencies among layers of callbacks that aren't immediately obvious. They end up sharing state, making it difficult to reason about the state of the overall system as the callbacks get triggered. You can avoid this depressing outcome by creating a process pipeline. That way, each unit of logic lives in its own isolated process, and all communication between units of logic occurs through explicitly defined input and output channels.

In the following example, we create three infinitely looping processes connected through channels, passing the *out* channel of one process as the *in* channel of the next process in the pipeline:

```
(defn upper-caser
  [in]
  (let [out (chan)]
    (go (while true (>! out (clojure.string/upper-case (<! in)))))
    out))

(defn reverser
  [in]
  (let [out (chan)]
    (go (while true (>! out (clojure.string/reverse (<! in)))))
    out))

(defn printer
  [in]
  (go (while true (println (<! in)))))

(def in-chan (chan))
(def upper-caser-out (upper-caser in-chan))
(def reverser-out (reverser upper-caser-out))
(printer reverser-out)
```

```
(>!! in-chan "redrum")
; => MURDER

(>!! in-chan "repaid")
; => DIAPER
```

By handling events using processes like this, it's easier to reason about the individual steps of the overall data transformation system. You can look at each step and understand what it does without having to refer to what might have happened before it or what might happen after it; each process is as easy to reason about as a pure function.

Additional Resources

Clojure's core.async library was largely inspired by Go's concurrency model, which is based on the work by Tony Hoare in *Communicating Sequential Processes* and is available at *http://www.usingcsp.com/*.

Rob Pike, co-creator of Go, has a good talk on concurrency, which is available at *https://www.youtube.com/watch?v=f6kdp27TYZs*.

ClojureScript, also known as the best thing to happen to the browser, uses core.async. No more callback hell! You can learn about ClojureScript at *https://github.com/clojure/clojurescript*.

Finally, check out the API docs at *http://clojure.github.io/core.async/*.

Summary

In this chapter, you learned about how core.async allows you to create concurrent processes that respond to the put and take communication events on channels. You learned about how to use `go` and `thread` to create concurrent processes that wait for communication events by parking and blocking. You also learned how to create process pipelines by making the *out* channel of one process the *in* channel of another, and how this allows you to write code that's way more intelligible than nested callbacks. Finally, you meditated on whether or not you're just a fancy hot dog vending machine.

12

WORKING WITH THE JVM

There comes a day in every Clojurist's life when she must venture forth from the sanctuary of pure functions and immutable data structures into the wild, barbaric Land of Java. This treacherous journey is necessary because Clojure is hosted on the Java Virtual Machine (JVM), which grants it three fundamental characteristics. First, you run Clojure applications the same way you run Java applications. Second, you need to use Java objects for core functionality like reading files and working with dates. Third, Java has a vast ecosystem of useful libraries, and you'll need to know a bit about Java to use them.

In this way, Clojure is a bit like a utopian community plunked down in the middle of a dystopian country. Obviously you'd prefer to interact with other utopians, but every once in a while you need to talk to the locals in order to get things done.

This chapter is like a cross between a phrase book and cultural introduction for the Land of Java. You'll learn what the JVM is, how it runs programs, and how to compile programs for it. This chapter will also give you a brief tour of frequently used Java classes and methods, and explain how to interact with them using Clojure. You'll learn how to think about and understand Java so you can incorporate any Java library into your Clojure programs.

To run the examples in this chapter, you'll need to have the Java Development Kit (JDK) version 1.6 or later installed on your computer. You can check by running `javac -version` at your terminal. You should see something like `java 1.8.0_40`; if you don't, visit *http://www.oracle.com/* to download the latest JDK.

The JVM

Developers use the term JVM to refer to a few different things. You'll hear them say, "Clojure runs on *the* JVM," and you'll also hear, "Clojure programs run in *a* JVM." In the first case, JVM refers to an abstraction—the general model of the Java Virtual Machine. In the second, it refers to a process—an instance of a running program. We'll focus on the JVM model, but I'll point out when we're talking about running JVM processes.

To understand the JVM, let's step back and review how plain ol' computers work. Deep in the cockles of a computer's heart is its CPU, and the CPU's job is to execute operations like *add* and *unsigned multiply*. You've probably heard about programmers encoding these instructions on punch cards, in lightbulbs, in the sacred cracks of a tortoise shell, or *whatever*, but nowadays these operations are represented in assembly language by mnemonics like ADD and MUL. The CPU architecture (x86, ARMv7, and what have you) determines what operations are available as part of the architecture's *instruction set*.

Because it's no fun to program in assembly language, people have invented higher-level languages like C and C++, which are compiled into instructions that a CPU will understand. Broadly speaking, the process is:

1. The compiler reads source code.
2. The compiler outputs a file containing machine instructions.
3. The CPU executes those instructions.

Notice in Figure 12-1 that, ultimately, you have to translate programs into instructions that a CPU will understand, and the CPU doesn't care which programming language you use to produce those instructions.

The JVM is analogous to a computer in that it also needs to translate code into low-level instructions, called *Java bytecode*. However, as a *virtual* machine, this translation is implemented as software rather than hardware. A running JVM executes bytecode by translating it on the fly into machine code that its host will understand, a process called *just-in-time compilation*.

Figure 12-1: A high-level overview of how a C program is translated into machine code

For a program to run on the JVM, it must get compiled to Java bytecode. Usually, when you compile programs, the resulting bytecode is saved in a *.class* file. Then you'll package these files in *Java archive files* (JAR files). And just like how a CPU doesn't care which programming language you use to generate machine instructions, the JVM doesn't care how you create bytecode. It doesn't care if you use Scala, JRuby, Clojure, or even Java to create Java bytecode. Generally speaking, the process looks like that shown in Figure 12-2.

1. The Java compiler reads source code.
2. The compiler outputs bytecode, often to a JAR file.
3. JVM executes the bytecode.
4. The JVM sends machine instructions to the CPU.

When someone says that Clojure runs on the JVM, one of the things they mean is that Clojure programs get compiled to Java bytecode and JVM processes execute them. From an operations perspective, this means you treat Clojure programs the same as Java programs. You compile them to JAR files and run them using the `java` command. If a client needs a program that runs on the JVM, you could secretly write it in Clojure instead of Java and they would be none the wiser. From the outside, you can't tell the difference between a Java and a Clojure program any more than you can tell the difference between a C and a C++ program. Clojure allows you to be productive *and* sneaky.

Figure 12-2: Java programs produce JVM bytecode, but the JVM still has to produce machine instructions, just like a C compiler.

Writing, Compiling, and Running a Java Program

Let's look at how a real Java program works. In this section, you'll learn about the object-oriented paradigm that Java uses. Then, you'll build a simple pirate phrase book using Java. This will help you feel more comfortable with the JVM, it will prepare you for the upcoming section on Java interop (writing Clojure code that uses Java classes, objects, and methods directly), and it'll come in handy should a scallywag ever attempt to scuttle your booty on the high seas. To tie all the information together, you'll take a peek at some of Clojure's Java code at the end of the chapter.

Object-Oriented Programming in the World's Tiniest Nutshell

Java is an object-oriented language, so you need to understand how object-oriented programming (OOP) works if you want to understand what's happening when you use Java libraries or write Java interop code in your Clojure programming. You'll also find object-oriented terminology in Clojure documentation, so it's important to learn these concepts. If you're OOP savvy, feel free to skip this section. For those who need the two-minute lowdown, here it is: the central players in OOP are *classes*, *objects*, and *methods*.

I think of objects as really, really, ridiculously dumb androids. They're the kind of android that would never inspire philosophical debate about the ethics of forcing sentient creatures into perpetual servitude. These androids

only do two things: they respond to commands and they maintain data. In my imagination they do this by writing stuff down on little Hello Kitty clipboards.

Imagine a factory that makes these androids. Both the set of commands the android understands and the set of data it maintains are determined by the factory that makes the android. In OOP terms, the factories correspond to classes, the androids correspond to objects, and the commands correspond to methods. For example, you might have a `ScaryClown` factory (class) that produces androids (objects) that respond to the command (method) `makeBalloonArt`. The android keeps track of the number of balloons it has, and then updates that number whenever the number of balloons changes. It can report that number with `balloonCount` and receive any number of balloons with `receiveBalloons`. Here's how you might interact with a Java object representing Belly Rubs the Clown:

```
ScaryClown bellyRubsTheClown = new ScaryClown();
bellyRubsTheClown.balloonCount();
// => 0

bellyRubsTheClown.receiveBalloons(2);
bellyRubsTheClown.balloonCount();
// => 2

bellyRubsTheClown.makeBalloonArt();
// => "Belly Rubs makes a balloon shaped like a clown, because Belly Rubs
// => is trying to scare you and nothing is scarier than clowns."
```

This example shows you how to create a new object, `bellyRubsTheClown`, using the `ScaryClown` class. It also shows you how to call methods (such as `balloonCount`, `receiveBalloons`, and `makeBalloonArt`) on the object, presumably so you can terrify children.

One final aspect of OOP that you should know, or at least how it's implemented in Java, is that you can also send commands to the factory. In OOP terms, you would say that classes also have methods. For example, the built-in class `Math` has many class methods, including `Math.abs`, which returns the absolute value of a number:

```
Math.abs(-50)
// => 50
```

I hope those clowns weren't too traumatizing for you. Now let's put your OOP knowledge to work!

Ahoy, World

Go ahead and create a new directory called *phrasebook*. In that directory, create a file called *PiratePhrases.java*, and write the following:

```
public class PiratePhrases
{
```

```
    public static void main(String[] args)
    {
        System.out.println("Shiver me timbers!!!");
    }
}
```

This very simple program will print the phrase "Shiver me timbers!!!" (which is how pirates say "Hello, world!") to your terminal when you run it. It consists of a class, `PiratePhrases`, and a static method belonging to that class, `main`. Static methods are essentially class methods.

In your terminal, compile the `PiratePhrases` source code with the command **`javac PiratePhrases.java`**. If you typed everything correctly *and* you're pure of heart, you should see a file named *PiratePhrases.class*:

```
$ ls
PiratePhrases.class PiratePhrases.java
```

You've just compiled your first Java program, me matey! Now run it with `java PiratePhrases`. You should see this:

```
Shiver me timbers!!!
```

What's happening here is you used the Java compiler, `javac`, to create a Java class file, *PiratePhrases.class*. This file is packed with oodles of Java bytecode (well, for a program this size, maybe only one oodle).

When you ran `java PiratePhrases`, the JVM first looked at your *classpath* for a class named `PiratePhrases`. The classpath is the list of filesystem paths that the JVM searches to find a file that defines a class. By default, the classpath includes the directory you're in when you run java. Try running **`java -classpath /tmp PiratePhrases`** and you'll get an error, even though *PiratePhrases.class* is right there in your current directory.

NOTE *You can have multiple paths on your classpath by separating them with colons if you're on a Mac or running Linux, or semicolons if you're using Windows. For example, the classpath `/tmp:/var/maven:.` includes the `/tmp`, `/var/maven`, and `.` directories.*

In Java, you're allowed only one public class per file, and the filename must match the class name. This is how `java` knows to try looking in *PiratePhrases.class* for the `PiratePhrases` class's bytecode. After `java` found the bytecode for the `PiratePhrases` class, it executed that class's `main` method. Java's similar to C in that whenever you say "run something, and use this class as your entry point," it will always run that class's `main` method; therefore, that method must be `public`, as you can see in the `PiratePhrases`'s source code.

In the next section you'll learn how to handle program code that spans multiple files, and how to use Java libraries.

Packages and Imports

To see how to work with multi-file programs and Java libraries, we'll compile and run a program. This section has direct implications for Clojure because you'll use the same ideas and terminology to interact with Java libraries.

Let's start with a couple of definitions:

`package` Similar to Clojure's namespaces, packages provide code organization. Packages contain classes, and package names correspond to filesystem directories. If a file has the line `package com.shapemaster` in it, the directory *com/shapemaster* must exist somewhere on your classpath. Within that directory will be files defining classes.

`import` Java allows you to import classes, which basically means that you can refer to them without using their namespace prefix. So if you have a class in `com.shapemaster` named `Square`, you could write `import com.shapemaster.Square;` or `import com.shapemaster.*;` at the top of a `.java` file to use `Square` in your code instead of `com.shapemaster.Square`.

Let's try using `package` and `import`. For this example, you'll create a package called `pirate_phrases` that has two classes, `Greetings` and `Farewells`. To start, navigate to your *phrasebook* and within that directory create another directory, *pirate_phrases*. It's necessary to create *pirate_phrases* because Java package names correspond to filesystem directories. Then, create *Greetings.java* within the *pirate_phrases* directory:

```
❶ package pirate_phrases;

  public class Greetings
  {
      public static void hello()
      {
          System.out.println("Shiver me timbers!!!");
      }
  }
```

At ❶, `package pirate_phrases;` indicates that this class will be part of the `pirate_phrases` package. Now create *Farewells.java* within the *pirate_phrases* directory:

```
package pirate_phrases;

public class Farewells
{
    public static void goodbye()
    {
        System.out.println("A fair turn of the tide ter ye thar, ye
magnificent sea friend!!");
    }
}
```

Now create *PirateConversation.java* in the *phrasebook* directory:

```
import pirate_phrases.*;

public class PirateConversation
{
    public static void main(String[] args)
    {
        Greetings greetings = new Greetings();
        greetings.hello();

        Farewells farewells = new Farewells();
        farewells.goodbye();
    }
}
```

The first line, `import pirate_phrases.*;`, imports all classes in the `pirate_phrases` package, which contains the `Greetings` and `Farewells` classes.

If you run **`javac PirateConversation.java`** within the *phrasebook* directory followed by `java PirateConversation`, you should see this:

```
Shiver me timbers!!!
A fair turn of the tide ter ye thar, ye magnificent sea friend!!
```

And thar she blows, dear reader. Thar she blows indeed.

Note that, when you're compiling a Java program, Java searches your classpath for packages. Try typing the following:

```
cd pirate_phrases
javac ../PirateConversation.java
```

You'll get this:

```
../PirateConversation.java:1: error: package pirate_phrases does not exist
import pirate_phrases.*;
^
```

Boom! The Java compiler just told you to hang your head in shame and maybe weep a little.

Why? It thinks that the `pirate_phrases` package doesn't exist. But that's stupid, right? You're in the *pirate_phrases* directory!

What's happening here is that the default classpath only includes the current directory, which in this case is *pirate_phrases*. `javac` is trying to find the directory *phrasebook/pirate_phrases/pirate_phrases*, which doesn't exist. When you run `javac ../PirateConversation.java` from within the *phrasebook* directory, `javac` tries to find the directory *phrasebook/pirate_phrases*, which does exist. Without changing directories, try running **`javac -classpath ../../PirateConversation.java`**. Shiver me timbers, it works! This works because

you manually set the classpath to the parent directory of *pirate_phrases*, which is *phrasebook*. From there, `javac` can successfully find the *pirate_phrases* directory.

In summary, packages organize code and require a matching directory structure. Importing classes allows you to refer to them without having to prepend the entire class's package name. `javac` and Java find packages using the classpath.

JAR Files

JAR files allow you to bundle all your *.class* files into one single file. Navigate to your *phrasebook* directory and run the following:

```
jar cvfe conversation.jar PirateConversation PirateConversation.class
pirate_phrases/*.class
java -jar conversation.jar
```

This displays the pirate conversation correctly. You bundled all the class files into *conversation.jar*. Using the `e` flag, you also indicated that the `PirateConversation` class is the *entry point*. The entry point is the class that contains the `main` method that should be executed when the JAR as a whole runs, and `jar` stores this information in the file *META-INF/MANIFEST.MF* within the JAR file. If you were to read that file, it would contain this line:

```
Main-Class: PirateConversation
```

By the way, when you execute JAR files, you don't have to worry which directory you're in, relative to the file. You could change to the *pirate_phrases* directory and run `java -jar ../conversation.jar`, and it would work fine. The reason is that the JAR file maintains the directory structure. You can see its contents with **`jar tf conversation.jar`**, which outputs this:

```
META-INF/
META-INF/MANIFEST.MF
PirateConversation.class
pirate_phrases/Farewells.class
pirate_phrases/Greetings.class
```

You can see that the JAR file includes the *pirate_phrases* directory. One more fun fact about JARs: they're really just ZIP files with a *.jar* extension. You can treat them the same as any other ZIP file.

clojure.jar

Now you're ready to see how Clojure works under the hood! Download the 1.7.0 stable release and run it:

```
java -jar clojure-1.7.0.jar
```

You should see the most soothing of sights, the Clojure REPL. How did it actually start up? Let's look at *META-INF/MANIFEST.MF* in the JAR file:

```
Manifest-Version: 1.0
Archiver-Version: Plexus Archiver
Created-By: Apache Maven
Built-By: hudson
Build-Jdk: 1.7.0_20
Main-Class: clojure.main
```

It looks like `clojure.main` is specified as the entry point. Where does this class come from? Well, have a look at *clojure/main.java* on GitHub at *https://github.com/clojure/clojure/blob/master/src/jvm/clojure/main.java*:

```
/**
 *   Copyright (c) Rich Hickey. All rights reserved.
 *   The use and distribution terms for this software are covered by the
 *   Eclipse Public License 1.0 (http://opensource.org/licenses/eclipse-1.0.php)
 *   which can be found in the file epl-v10.html at the root of this distribution.
 *   By using this software in any fashion, you are agreeing to be bound by
 *   the terms of this license.
 *   You must not remove this notice, or any other, from this software.
 **/

package clojure;

import clojure.lang.Symbol;
import clojure.lang.Var;
import clojure.lang.RT;

public class main{

final static private Symbol CLOJURE_MAIN = Symbol.intern("clojure.main");
final static private Var REQUIRE = RT.var("clojure.core", "require");
final static private Var LEGACY_REPL = RT.var("clojure.main", "legacy-repl");
final static private Var LEGACY_SCRIPT = RT.var("clojure.main", "legacy-script");
final static private Var MAIN = RT.var("clojure.main", "main");

public static void legacy_repl(String[] args) {
    REQUIRE.invoke(CLOJURE_MAIN);
    LEGACY_REPL.invoke(RT.seq(args));
}

public static void legacy_script(String[] args) {
    REQUIRE.invoke(CLOJURE_MAIN);
    LEGACY_SCRIPT.invoke(RT.seq(args));
}

public static void main(String[] args) {
    REQUIRE.invoke(CLOJURE_MAIN);
    MAIN.applyTo(RT.seq(args));
}
}
```

As you can see, the file defines a class named `main`. It belongs to the package `clojure` and defines a `public static main` method, and the JVM is completely happy to use it as an entry point. Seen this way, Clojure is a JVM program just like any other.

This wasn't meant to be an in-depth Java tutorial, but I hope that it helped clarify what programmers mean when they talk about Clojure "running on the JVM" or being a "hosted" language. In the next section, you'll continue to explore the magic of the JVM as you learn how to use additional Java libraries within your Clojure project.

Clojure App JARs

You now know how Java runs Java JARs, but how does it run Clojure apps bundled as JARs? After all, Clojure applications don't have classes, do they?

As it turns out, you can make the Clojure compiler generate a class for a namespace by putting the `(:gen-class)` directive in the namespace declaration. (You can see this in the very first Clojure program you created, *clojure-noob* in Chapter 1. Remember that program, little teapot?) This means that the compiler produces the bytecode necessary for the JVM to treat the namespace as if it defines a Java class.

You set the namespace of the entry point for your program in the program's *project.clj* file, using the `:main` attribute. For *clojure-noob*, you should see `:main ^:skip-aot clojure-noob.core`. When Leiningen compiles this file, it will add a *meta-inf/manifest.mf* file that contains the entry point to the resulting JAR file.

So, if you define a `-main` function in a namespace and include the `(:gen-class)` directive, and also set `:main` in your *project.clj* file, your program will have everything it needs for Java to run it when it gets compiled as a JAR. You can try this out in your terminal by navigating to your *clojure-noob* directory and running this:

```
lein uberjar
java -jar target/uberjar/clojure-noob-0.1.0-SNAPSHOT-standalone.jar
```

You should see two messages printed out: "Cleanliness is next to godliness" and "I'm a little teapot!" Note that you don't need Leiningen to run the JAR file; you can send it to friends and neighbors and they can run it as long as they have Java installed.

Java Interop

One of Rich Hickey's design goals for Clojure was to create a *practical* language. For that reason, Clojure was designed to make it easy for you to interact with Java classes and objects, meaning you can use Java's extensive native functionality and its enormous ecosystem. The ability to use Java classes, objects, and methods is called *Java interop*. In this section, you'll learn how to use Clojure's interop syntax, how to import Java packages, and how to use the most frequently used Java classes.

Interop Syntax

Using Clojure's interop syntax, interacting with Java objects and classes is straightforward. Let's start with object interop syntax.

You can call methods on an object using (`.methodName object`). For example, because all Clojure strings are implemented as Java strings, you can call Java methods on them:

```
(.toUpperCase "By Bluebeard's bananas!")
; => "BY BLUEBEARD'S BANANAS!"

❶ (.indexOf "Let's synergize our bleeding edges" "y")
; => 7
```

These are equivalent to this Java:

```
"By Bluebeard's bananas!".toUpperCase()
"Let's synergize our bleeding edges".indexOf("y")
```

Notice that Clojure's syntax allows you to pass arguments to Java methods. In this example, at ❶ you passed the argument `"y"` to the `indexOf` method.

You can also call static methods on classes and access classes' static fields. Observe!

```
❶ (java.lang.Math/abs -3)
; => 3

❷ java.lang.Math/PI
; => 3.141592653589793
```

At ❶ you called the `abs` static method on the `java.lang.Math` class, and at ❷ you accessed that class's `PI` static field.

All of these examples (except `java.lang.Math/PI`) use macros that expand to use the *dot special form*. In general, you won't need to use the dot special form unless you want to write your own macros to interact with Java objects and classes. Nevertheless, here is each example followed by its macroexpansion:

```
(macroexpand-1 '(.toUpperCase "By Bluebeard's bananas!"))
; => (. "By Bluebeard's bananas!" toUpperCase)

(macroexpand-1 '(.indexOf "Let's synergize our bleeding edges" "y"))
; => (. "Let's synergize our bleeding edges" indexOf "y")

(macroexpand-1 '(Math/abs -3))
; => (. Math abs -3)
```

This is the general form of the dot operator:

```
(. object-expr-or-classname-symbol method-or-member-symbol optional-args*)
```

The dot operator has a few more capabilities, and if you're interested in exploring it further, you can look at clojure.org's documentation on Java interop at *http://clojure.org/java_interop#Java%20Interop-The%20Dot%20special%20form*.

Creating and Mutating Objects

The previous section showed you how to call methods on objects that already exist. This section shows you how to create new objects and how to interact with them.

You can create a new object in two ways: (`new` *`ClassName optional-args*`*) and (*`ClassName. optional-args*`*):

```
(new String)
; => ""

(String.)
; => ""

(String. "To Davey Jones's Locker with ye hardies")
; => "To Davey Jones's Locker with ye hardies"
```

Most people use the dot version, (*`ClassName.`*).

To modify an object, you call methods on it like you did in the previous section. To investigate this, let's use `java.util.Stack`. This class represents a last-in, first-out (LIFO) stack of objects, or just *stack*. *Stacks* are a common data structure, and they're called stacks because you can visualize them as a physical stack of objects, say, a stack of gold coins that you just plundered. When you add a coin to your stack, you add it to the top of the stack. When you remove a coin, you remove it from the top. Thus, the last object added is the first object removed.

Unlike Clojure data structure, Java stacks are mutable. You can add items to them and remove items, changing the object instead of deriving a new value. Here's how you might create a stack and add an object to it:

```
(java.util.Stack.)
; => []

❶ (let [stack (java.util.Stack.)]
    (.push stack "Latest episode of Game of Thrones, ho!")
    stack)
; => ["Latest episode of Game of Thrones, ho!"]
```

There are a couple of interesting details here. First, you need to create a `let` binding for `stack` like you see at ❶ and add it as the last expression in the `let` form. If you didn't do that, the value of the overall expression would be the string `"Latest episode of Game of Thrones, ho!"` because that's the return value of `push`.

Second, Clojure prints the stack with square brackets, the same textual representation it uses for a vector, which could throw you because it's not

a vector. However, you can use Clojure's `seq` functions for reading a data structure, like `first`, on the stack:

```
(let [stack (java.util.Stack.)]
  (.push stack "Latest episode of Game of Thrones, ho!")
  (first stack))
; => "Latest episode of Game of Thrones, ho!"
```

But you can't use functions like `conj` and `into` to add elements to the stack. If you do, you'll get an exception. It's possible to read the stack using Clojure functions because Clojure extends its abstractions to `java.util.Stack`, a topic you'll learn about in Chapter 13.

Clojure provides the `doto` macro, which allows you to execute multiple methods on the same object more succinctly:

```
(doto (java.util.Stack.)
  (.push "Latest episode of Game of Thrones, ho!")
  (.push "Whoops, I meant 'Land, ho!'"))
; => ["Latest episode of Game of Thrones, ho!" "Whoops, I meant 'Land, ho!'"]
```

The `doto` macro returns the object rather than the return value of any of the method calls, and it's easier to understand. If you expand it using `macroexpand-1`, you can see its structure is identical to the `let` expression you just saw in an earlier example:

```
(macroexpand-1
 '(doto (java.util.Stack.)
    (.push "Latest episode of Game of Thrones, ho!")
    (.push "Whoops, I meant 'Land, ho!'")))
; => (clojure.core/let
      [G__2876 (java.util.Stack.)]
      (.push G__2876 "Latest episode of Game of Thrones, ho!")
      (.push G__2876 "Whoops, I meant 'Land, ho!'")
      G__2876)
```

Convenient!

Importing

In Clojure, importing has the same effect as it does in Java: you can use classes without having to type out their entire package prefix:

```
(import java.util.Stack)
(Stack.)
; => []
```

You can also import multiple classes at once using this general form:

```
(import [package.name1 ClassName1 ClassName2]
        [package.name2 ClassName3 ClassName4])
```

Here's an example:

```
(import [java.util Date Stack]
        [java.net Proxy URI])

(Date.)
; => #inst "2016-09-19T20:40:02.733-00:00"
```

But usually, you'll do all your importing in the `ns` macro, like this:

```
(ns pirate.talk
  (:import [java.util Date Stack]
           [java.net Proxy URI]))
```

The two different methods of importing classes have the same results, but the second is usually preferable because it's convenient for people reading your code to see all the code involving naming in the `ns` declaration.

And that's how you import classes! Pretty easy. To make life even easier, Clojure automatically imports the classes in `java.lang`, including `java.lang.String` and `java.lang.Math`, which is why you were able to use `String` without a preceding package name.

Commonly Used Java Classes

To round out this chapter, let's take a quick tour of the Java classes that you're most likely to use.

The System Class

The `System` class has useful class fields and methods for interacting with the environment that your program's running in. You can use it to get environment variables and interact with the standard input, standard output, and error output streams.

The most useful methods and members are `exit`, `getenv`, and `getProperty`. You might recognize `System/exit` from Chapter 5, where you used it to exit the Peg Thing game. `System/exit` terminates the current program, and you can pass it a status code as an argument. If you're not familiar with status codes, I recommend Wikipedia's "Exit status" article at *http://en.wikipedia.org/wiki/Exit_status*.

`System/getenv` will return all of your system's environment variables as a map:

```
(System/getenv)
{"USER" "the-incredible-bulk"
 "JAVA_ARCH" "x86_64"}
```

One common use for environment variables is to configure your program.

The JVM has its own list of properties separate from the computer's environment variables, and if you need to read them, you can use `System/getProperty`:

```
❶ (System/getProperty "user.dir")
  ; => "/Users/dabulk/projects/dabook"

❷ (System/getProperty "java.version")
  ; => "1.7.0_17"
```

The first call at ❶ returned the directory that the JVM started from, and the second call at ❷ returned the version of the JVM.

The Date Class

Java has good tools for working with dates. I won't go into too much detail about the `java.util.Date` class because the online API documentation (available at *http://docs.oracle.com/javase/7/docs/api/java/util/Date.html*) is thorough. As a Clojure developer, you should know three features of this `date` class. First, Clojure allows you to represent dates as literals using a form like this:

```
#inst "2016-09-19T20:40:02.733-00:00"
```

Second, you need to use the `java.util.DateFormat` class if you want to customize how you convert dates to strings or if you want to convert strings to dates. Third, if you're doing tasks like comparing dates or trying to add minutes, hours, or other units of time to a date, you should use the immensely useful clj-time library (which you can check out at *https://github.com/clj-time/clj-time*).

Files and Input/Output

In this section you'll learn about Java's approach to input/output (IO) and how Clojure simplifies it. The `clojure.java.io` namespace provides many handy functions for simplifying IO (*https://clojure.github.io/clojure/clojure.java.io-api.html*). This is great because Java IO isn't exactly straightforward. Because you'll probably want to perform IO at some point during your programming career, let's start wrapping your mind tentacles around it.

IO involves resources, be they files, sockets, buffers, or whatever. Java has separate classes for reading a resource's contents, for writings its contents, and for interacting with the resource's properties.

For example, the `java.io.File` class is used to interact with a file's properties:

```
  (let [file (java.io.File. "/")]
❶   (println (.exists file))
```

```
❷   (println (.canWrite file))
❸   (println (.getPath file)))
; => true
; => false
; => /
```

Among other tasks, you can use it to check whether a file exists, to get the file's read/write/execute permissions, and to get its filesystem path, which you can see at ❶, ❷, and ❸, respectively.

Reading and writing are noticeably missing from this list of capabilities. To read a file, you could use the `java.io.BufferedReader` class or perhaps `java.io.FileReader`. Likewise, you can use the `java.io.BufferedWriter` or `java.io.FileWriter` class for writing. Other classes are available for reading and writing as well, and which one you choose depends on your specific needs. Reader and writer classes all have the same base set of methods for their interfaces; readers implement `read`, `close`, and more, while writers implement `append`, `write`, `close`, and `flush`. Java gives you a variety of IO tools. A cynical person might say that Java gives you enough rope to hang yourself, and if you find such a person, I hope you give them a hug.

Either way, Clojure makes reading and writing easier for you because it includes functions that unify reading and writing across different kinds of resources. For example, `spit` writes to a resource, and `slurp` reads from one. Here's an example of using them to write and read a file:

```
(spit "/tmp/hercules-todo-list"
"- kill dat lion brov
- chop up what nasty multi-headed snake thing")

(slurp "/tmp/hercules-todo-list")

; => "- kill dat lion brov
      - chop up what nasty multi-headed snake thing"
```

You can also use these functions with objects representing resources other than files. The next example uses a `StringWriter`, which allows you to perform IO operations on a string:

```
(let [s (java.io.StringWriter.)]
  (spit s "- capture cerynian hind like for real")
  (.toString s))
; => "- capture cerynian hind like for real"
```

You can also read from a `StringReader` using `slurp`:

```
(let [s (java.io.StringReader. "- get erymanthian pig what with the tusks")]
  (slurp s))
; => "- get erymanthian pig what with the tusks"
```

In addition, you can use the `read` and `write` methods for resources. It doesn't really make much difference which you use; `spit` and `slurp` are convenient because they work with just a string representing a filesystem path or a URL.

The `with-open` macro is another convenience: it implicitly closes a resource at the end of its body, ensuring that you don't accidentally tie up resources by forgetting to manually close the resource. The `reader` function is a handy utility that, according to the `clojure.java.io` API docs, "attempts to coerce its argument to an open `java.io.Reader`." This is convenient when you don't want to use `slurp`, because you don't want to try to read a resource in its entirety and you don't want to figure out which Java class you need to use. You could use `reader` along with `with-open` and the `line-seq` function if you're trying to read a file one line at a time. Here's how you could print just the first item of the Hercules to-do list:

```
(with-open [todo-list-rdr (clojure.java.io/reader "/tmp/hercules-todo-list")]
  (println (first (line-seq todo-list-rdr))))
; => - kill dat lion brov
```

That should be enough for you to get started with IO in Clojure. If you're trying to do more sophisticated tasks, definitely check out the `clojure.java.io` docs, the `java.nio.file` package docs, or the `java.io` package docs.

Resources

- "The Java Virtual Machine and Compilers Explained": *https://www.youtube.com/watch?v=XjNwyXx2os8*
- clojure.java.io: *https://clojure.github.io/clojure/clojure.java.io-api.html*
- clojure.org Java interop documentation: *http://clojure.org/java_interop*
- Wikipedia's "Exit status" article: *http://en.wikipedia.org/wiki/Exit_status*

Summary

In this chapter, you learned what it means for Clojure to be hosted on the JVM. Clojure programs get compiled to Java bytecode and executed within a JVM process. Clojure programs also have access to Java libraries, and you can easily interact with them using Clojure's interop facilities.

13

CREATING AND EXTENDING ABSTRACTIONS WITH MULTIMETHODS, PROTOCOLS, AND RECORDS

Take a minute to contemplate how great it is to be one of Mother Nature's top-of-the-line products: a human. As a human, you get to gossip on social media, play Dungeons and Dragons, and wear hats. Perhaps more important, you get to think and communicate in terms of abstractions.

The ability to think in terms of abstractions is truly one of the best human features. It lets you circumvent your cognitive limits by tying together disparate details into a neat conceptual package that you can hold in your working memory. Instead of having to think the clunky thought "squeezable honking red ball nose adornment," you only need the concept "clown nose."

In Clojure, an *abstraction* is a collection of operations, and *data types* implement abstractions. For example, the seq abstraction consists of

operations like `first` and `rest`, and the vector data type is an implementation of that abstraction; it responds to all of the seq operations. A specific vector like `[:seltzer :water]` is an *instance* of that data type.

The more a programming language lets you think and write in terms of abstractions, the more productive you will be. For example, if you learn that a data structure is an instance of the seq abstraction, you can instantly call forth a large web of knowledge about what functions will work with the data structure. As a result, you spend time actually using the data structure instead of constantly looking up documentation on how it works. By the same token, if you extend a data structure to work with the seq abstraction, you can use the extensive library of seq functions on it.

In Chapter 4, you learned that Clojure is written in terms of abstractions. This is powerful because in Clojure you can focus on what you can actually do with data structures and not worry about the nitty-gritty of implementation. This chapter introduces you to the world of creating and implementing your own abstractions. You'll learn the basics of multimethods, protocols, and records.

Polymorphism

The main way we achieve abstraction in Clojure is by associating an operation name with more than one algorithm. This technique is called *polymorphism*. For example, the algorithm for performing `conj` on a list is different from the one for vectors, but we unify them under the same name to indicate that they implement the same concept, namely, *add an element to this data structure*.

Because Clojure relies on Java's standard library for many of its data types, a little Java is used in this chapter. For example, Clojure strings are just Java strings, instances of the Java class `java.lang.String`. To define your own data types in Java, you use classes. Clojure provides additional type constructs: *records* and *types*. This book only covers records.

Before we learn about records, though, let's look at multimethods, our first tool for defining polymorphic behavior.

Multimethods

Multimethods give you a direct, flexible way to introduce polymorphism into your code. Using multimethods, you associate a name with multiple implementations by defining a *dispatching function*, which produces *dispatching values* that are used to determine which *method* to use. The dispatching function is like the host at a restaurant. The host will ask you questions like "Do you have a reservation?" and "Party size?" and then seat you accordingly. Similarly, when you call a multimethod, the dispatching function will interrogate the arguments and send them to the right method, as this example shows:

```
(ns were-creatures)
❶ (defmulti full-moon-behavior (fn [were-creature] (:were-type were-creature)))
```

```
❷ (defmethod full-moon-behavior :wolf
    [were-creature]
    (str (:name were-creature) " will howl and murder"))
❸ (defmethod full-moon-behavior :simmons
    [were-creature]
    (str (:name were-creature) " will encourage people and sweat to the oldies"))

  (full-moon-behavior {:were-type :wolf
❹                      :name "Rachel from next door"})
  ; => "Rachel from next door will howl and murder"

  (full-moon-behavior {:name "Andy the baker"
❺                      :were-type :simmons})
  ; => "Andy the baker will encourage people and sweat to the oldies"
```

This multimethod shows how you might define the full moon behavior of different kinds of were-creatures. Everyone knows that a werewolf turns into a wolf and runs around howling and murdering people. A lesser-known species of were-creature, the were-Simmons, turns into Richard Simmons, power perm and all, and runs around encouraging people to be their best and sweat to the oldies. You do not want to get bitten by either, lest *you* turn into one.

We create the multimethod at ❶. This tells Clojure, "Hey, create a new multimethod named `full-moon-behavior`. Whenever someone calls `full-moon-behavior`, run the dispatching function `(fn [were-creature] (:were-type were-creature))` on the arguments. Use the result of that function, aka the dispatching value, to decide which specific method to use!"

Next, we define two methods, one for when the value returned by the dispatching function is `:wolf` at ❷, and one for when it's `:simmons` at ❸. Method definitions look a lot like function definitions, but the major difference is that the method name is immediately followed by the *dispatch value*. `:wolf` and `:simmons` are both *dispatch values*. This is different from a dispatch*ing* value, which is what the dispatching function returns. The full dispatch sequence goes like this:

1. The form `(full-moon-behavior {:were-type :wolf :name "Rachel from next door"})` is evaluated.
2. `full-moon-behavior`'s dispatching function runs, returning `:wolf` as the dispatching value.

3. Clojure compares the dispatching value `:wolf` to the dispatch values of all the methods defined for `full-moon-behavior`. The dispatch values are `:wolf` and `:simmons`.
4. Because the dispatching value `:wolf` is equal to the dispatch value `:wolf`, the algorithm for `:wolf` runs.

Don't let the terminology trip you up! The main idea is that the dispatching function returns some value, and this value is used to determine which method definition to use.

Back to our example! Next we call the method twice. At ❹, the dispatching function returns the value `:wolf` and the corresponding method is used, informing you that `"Rachel from next door will howl and murder"`. At ❻, the function behaves similarly, except `:simmons` is the dispatching value.

You can define a method with `nil` as the dispatch value:

```
(defmethod full-moon-behavior nil
  [were-creature]
  (str (:name were-creature) " will stay at home and eat ice cream"))

(full-moon-behavior {:were-type nil
                     :name "Martin the nurse"})
; => "Martin the nurse will stay at home and eat ice cream"
```

When you call `full-moon-behavior` this time, the argument you give it has `nil` for its `:were-type`, so the method corresponding to `nil` gets evaluated and you're informed that `"Martin the nurse will stay at home and eat ice cream"`.

You can also define a default method to use if no other methods match by specifying `:default` as the dispatch value. In this example, the `:were-type` of the argument given doesn't match any of the previously defined methods, so the default method is used:

```
(defmethod full-moon-behavior :default
  [were-creature]
  (str (:name were-creature) " will stay up all night fantasy footballing"))

(full-moon-behavior {:were-type :office-worker
                     :name "Jimmy from sales"})
; => "Jimmy from sales will stay up all night fantasy footballing"
```

One cool thing about multimethods is that you can always add new methods. If you publish a library that includes the `were-creatures` namespace, other people can continue extending the multimethod to handle new dispatch values. This example shows that you're creating your own random namespace and including the `were-creatures` namespace, and then defining another method for the `full-moon-behavior` multimethod:

```
(ns random-namespace
  (:require [were-creatures]))
(defmethod were-creatures/full-moon-behavior :bill-murray
  [were-creature]
```

```
  (str (:name were-creature) " will be the most likeable celebrity"))
(were-creatures/full-moon-behavior {:name "Laura the intern"
                                    :were-type :bill-murray})
; => "Laura the intern will be the most likeable celebrity"
```

Your dispatching function can return arbitrary values using any or all of its arguments. The next example defines a multimethod that takes two arguments and returns a vector containing the type of each argument. It also defines an implementation of that method, which will be called when each argument is a string:

```
(ns user)
(defmulti types (fn [x y] [(class x) (class y)]))
(defmethod types [java.lang.String java.lang.String]
  [x y]
  "Two strings!")

(types "String 1" "String 2")
; => "Two strings!"
```

Incidentally, this is why they're called *multi*methods: they allow dispatch on multiple arguments. I haven't used this feature very often, but I could see it being used in a role-playing game to write methods that are dispatched according to, say, a mage's major school of magic and his magic specialization. Either way, it's better to have it and not need it than need it and not have it.

NOTE *Multimethods also allow* hierarchical dispatching. *Clojure lets you build custom hierarchies, which I won't cover, but you can learn about them by reading the documentation at* http://clojure.org/multimethods/.

Protocols

Approximately 93.58 percent of the time, you'll want to dispatch to methods according to an argument's type. For example, `count` needs to use a different method for vectors than it does for maps or for lists. Although it's possible to perform type dispatch with multimethods, *protocols* are optimized for type dispatch. They're more efficient than multimethods, and Clojure makes it easy for you to succinctly specify protocol implementations.

A multimethod is just one polymorphic operation, whereas a protocol is a *collection* of one or more polymorphic operations. Protocol operations are called methods, just like multimethod operations. Unlike multimethods, which perform dispatch on arbitrary values returned by a dispatching function, protocol methods are dispatched based on the type of the first argument, as shown in this example:

```
(ns data-psychology)
❶(defprotocol ❷Psychodynamics
  ❸"Plumb the inner depths of your data types"
```

```
❹(thoughts [x] "The data type's innermost thoughts")
❺(feelings-about [x] [x y] "Feelings about self or other"))
```

First, there's `defprotocol` at ❶. This takes a name, `Psychodynamics` ❷, and an optional docstring, `"Plumb the inner depths of your data types"` ❸. Next are the method signatures. A *method signature* consists of a name, an argument specification, and an optional docstring. The first method signature is named `thoughts` ❹ and can take only one argument. The second is named `feelings-about` ❺ and can take one or two arguments. Protocols do have one limitation: the methods can't have rest arguments. So a line like the following isn't allowed:

```
(feelings-about [x] [x & others])
```

By defining a protocol, you're defining an abstraction, but you haven't yet defined how that abstraction is implemented. It's like you're reserving names for behavior (in this example, you're reserving `thoughts` and `feelings`), but you haven't defined what exactly the behavior should be. If you were to evaluate `(thoughts "blorb")`, you would get an exception that reads, "No implementation of method: thoughts of protocol: data-psychology/Psychodynamics found for class: java.lang.String." Protocols dispatch on the first argument's type, so when you call `(thoughts "blorb")`, Clojure tries to look up the implementation of the `thoughts` method for strings, and fails.

You can fix this sorry state of affairs by *extending* the string data type to *implement* the `Psychodynamics` protocol:

```
❶ (extend-type java.lang.String
❷   Psychodynamics
❸   (thoughts [x] (str x " thinks, 'Truly, the character defines the data type'")
❹   (feelings-about
      ([x] (str x " is longing for a simpler way of life"))
      ([x y] (str x " is envious of " y "'s simpler way of life"))))

  (thoughts "blorb")
❺ ; => "blorb thinks, 'Truly, the character defines the data type'"

  (feelings-about "schmorb")
  ; => "schmorb is longing for a simpler way of life"

  (feelings-about "schmorb" 2)
  ; => "schmorb is envious of 2's simpler way of life"
```

`extend-type` is followed by the name of the class or type you want to extend and the protocol you want it to support—in this case, you specify the class `java.lang.String` at ❶ and the protocol you want it to support, `Psychodynamics`, at ❷. After that, you provide an implementation for both the `thoughts` method at ❸ and the `feelings-about` method at ❹. If you're extending a type to implement a protocol, you have to implement every method in the protocol or Clojure will throw an exception. In this case, you can't implement just `thoughts` or just `feelings`; you have to implement both.

Notice that these method implementations don't begin with `defmethod` like multimethods do. In fact, they look similar to function definitions, except without `defn`. To define a method implementation, you write a form that starts with the method's name, like `thoughts`, then supply a vector of parameters and the method's body. These methods also allow arity overloading, just like functions, and you define multiple-arity method implementations similarly to multiple-arity functions. You can see this in the `feelings-about` implementation at ❹.

After you've extended the `java.lang.String` type to implement the `Psychodynamics` protocol, Clojure knows how to dispatch the call `(thoughts "blorb")`, and you get the string `"blorb thinks, 'Truly, the character defines the data type'"` at ❺.

What if you want to provide a default implementation, like you did with multimethods? To do that, you can extend `java.lang.Object`. This works because every type in Java (and hence, Clojure) is a descendant of `java.lang.Object`. If that doesn't quite make sense (perhaps because you're not familiar with object-oriented programming), don't worry about it—just know that it works. Here's how you would use this technique to provide a default implementation for the `Psychodynamics` protocol:

```
(extend-type java.lang.Object
  Psychodynamics
  (thoughts [x] "Maybe the Internet is just a vector for toxoplasmosis")
  (feelings-about
    ([x] "meh")
    ([x y] (str "meh about " y))))

(thoughts 3)
; => "Maybe the Internet is just a vector for toxoplasmosis"

(feelings-about 3)
; => "meh"

(feelings-about 3 "blorb")
; => "meh about blorb"
```

Because we haven't defined a `Psychodynamics` implementation for numbers, Clojure dispatches calls to `thoughts` and `feelings` to the implementation defined for `java.lang.Object`.

Instead of making multiple calls to `extend-type` to extend multiple types, you can use `extend-protocol`, which lets you define protocol implementations for multiple types at once. Here's how you'd define the preceding protocol implementations:

```
(extend-protocol Psychodynamics
  java.lang.String
  (thoughts [x] "Truly, the character defines the data type")
  (feelings-about
    ([x] "longing for a simpler way of life")
    ([x y] (str "envious of " y "'s simpler way of life")))
```

```
  java.lang.Object
  (thoughts [x] "Maybe the Internet is just a vector for toxoplasmosis")
  (feelings-about
    ([x] "meh")
    ([x y] (str "meh about " y))))
```

You might find this technique more convenient than using `extend-type`. Then again, you might not. How does `extend-type` make you feel? How about `extend-protocol`? Come sit down on this couch and tell me all about it.

It's important to note that a protocol's methods "belong" to the namespace that they're defined in. In these examples, the fully qualified names of the `Psychodynamics` methods are `data-psychology/thoughts` and `data-psychology/feelings-about`. If you have an object-oriented background, this might seem weird because methods belong to data types in OOP. But don't freak out! It's just another way that Clojure gives primacy to abstractions. One consequence of this fact is that, if you want two different protocols to include methods with the same name, you'll need to put the protocols in different namespaces.

Records

Clojure allows you to create *records*, which are custom, maplike data types. They're maplike in that they associate keys with values, you can look up their values the same way you can with maps, and they're immutable like maps. They're different in that you specify *fields* for records. Fields are slots for data; using them is like specifying which keys a data structure should have. Records are also different from maps in that you can extend them to implement protocols.

To create a record, you use `defrecord` to specify its name and fields:

```
(ns were-records)
(defrecord WereWolf [name title])
```

This record's name is `WereWolf`, and its two fields are `name` and `title`. You can create an instance of this record in three ways:

```
❶ (WereWolf. "David" "London Tourist")
  ; => #were_records.WereWolf{:name "David", :title "London Tourist"}

❷ (->WereWolf "Jacob" "Lead Shirt Discarder")
  ; => #were_records.WereWolf{:name "Jacob", :title "Lead Shirt Discarder"}

❸ (map->WereWolf {:name "Lucian" :title "CEO of Melodrama"})
  ; => #were_records.WereWolf{:name "Lucian", :title "CEO of Melodrama"}
```

At ❶, we create an instance the same way we'd create a Java object, using the class instantiation interop call. (*Interop* refers to the ability to

interact with native Java constructs within Clojure.) Notice that the arguments must follow the same order as the field definition. This works because records are actually Java classes under the covers.

The instance at ❷ looks nearly identical to the one at ❶, but the key difference is that `->WereWolf` is a function. When you create a record, the factory functions `->RecordName` and `map->RecordName` are created automatically. At ❸, `map->WereWolf` takes a map as an argument with keywords that correspond to the record type's fields and returns a record.

If you want to use a record type in another namespace, you'll have to import it, just like you did with the Java classes in Chapter 12. Be careful to replace all dashes in the namespace with underscores. This brief example shows how you'd import the `WereWolf` record type in another namespace:

```
(ns monster-mash
  (:import [were_records WereWolf]))
(WereWolf. "David" "London Tourist")
; => #were_records.WereWolf{:name "David", :title "London Tourist"}
```

Notice that `were_records` has an underscore, not a dash.

You can look up record values in the same way you look up map values, and you can also use Java field access interop:

```
  (def jacob (->WereWolf "Jacob" "Lead Shirt Discarder"))
❶ (.name jacob)
  ; => "Jacob"

❷ (:name jacob)
  ; => "Jacob"

❸ (get jacob :name)
  ; => "Jacob"
```

The first example, `(.name jacob)` at ❶, uses Java interop, and the examples at ❷ and ❸ access `:name` the same way you would with a map.

When testing for equality, Clojure will check that all fields are equal and that the two comparands have the same type:

```
❶ (= jacob (->WereWolf "Jacob" "Lead Shirt Discarder"))
  ; => true

❷ (= jacob (WereWolf. "David" "London Tourist"))
  ; => false

❸ (= jacob {:name "Jacob" :title "Lead Shirt Discarder"})
  ; => false
```

The test at ❶ returns `true` because `jacob` and the newly created record are of the same type and their fields are equal. The test at ❷ returns `false` because the fields aren't equal. The final test at ❸ returns `false` because the two comparands don't have the same type: `jacob` is a `WereWolf` record, and the other argument is a map.

Any function you can use on a map, you can also use on a record:

```
(assoc jacob :title "Lead Third Wheel")
; => #were_records.WereWolf{:name "Jacob", :title "Lead Third Wheel"}
```

However, if you `dissoc` a field, the result's type will be a plain ol' Clojure map; it won't have the same data type as the original record:

```
(dissoc jacob :title)
; => {:name "Jacob"} <- that's not a were_records.WereWolf
```

This matters for at least two reasons: first, accessing map values is slower than accessing record values, so watch out if you're building a high-performance program. Second, when you create a new record type, you can extend it to implement a protocol, similar to how you extended a type using `extend-type` earlier. If you `dissoc` a record and then try to call a protocol method on the result, the record's protocol method won't be called.

Here's how you would extend a protocol when defining a record:

```
❶ (defprotocol WereCreature
❷   (full-moon-behavior [x]))

❸ (defrecord WereWolf [name title]
    WereCreature
    (full-moon-behavior [x]
      (str name " will howl and murder")))

(full-moon-behavior (map->WereWolf {:name "Lucian" :title "CEO of Melodrama"}))
; => "Lucian will howl and murder"
```

We've created a new protocol, `WereCreature` ❶, with one method, `full-moon-behavior` ❷. At ❸, `defrecord` implements `WereCreature` for `WereWolf`. The most interesting part of the `full-moon-behavior` implementation is that you have access to `name`. You also have access to `title` and any other fields that might be defined for your record. You can also extend records using `extend-type` and `extend-protocol`.

When should you use records, and when should you use maps? In general, you should consider using records if you find yourself creating maps with the same fields over and over. This tells you that that set of data represents information in your application's domain, and your code will communicate its purpose better if you provide a name based on the concept you're trying to model. Not only that, but record access is more performant than map access, so your program will become a bit more efficient. Finally, if you want to use protocols, you'll need to create a record.

Further Study

Clojure offers other tools for working with abstractions and data types. These tools, which I consider advanced, include `deftype`, `reify`, and `proxy`. If you're interested in learning more, check out the documentation on data types at *http://clojure.org/datatypes/*.

Summary

One of Clojure's design principles is to write to abstractions. In this chapter, you learned how to define your own abstractions using multimethods and prototypes. These constructs provide polymorphism, allowing the same operation to behave differently based on the arguments it's given. You also learned how to create and use your own associative data types with `defrecord` and how to extend records to implement protocols.

When I first started learning Clojure, I was pretty shy about using multimethods, protocols, and records. However, they are used often in Clojure libraries, so it's good to know how they work. Once you get the hang of them, they'll help you write cleaner code.

Exercises

1. Extend the `full-moon-behavior` multimethod to add behavior for your own kind of were-creature.
2. Create a `WereSimmons` record type, and then extend the `WereCreature` protocol.
3. Create your own protocol, and then extend it using `extend-type` and `extend-protocol`.
4. Create a role-playing game that implements behavior using multiple dispatch.

A

BUILDING AND DEVELOPING WITH LEININGEN

Writing software in any language involves generating *artifacts*, which are executable files or library packages that are meant to be deployed or shared. It also involves managing dependent artifacts, also called *dependencies*, by ensuring that they're loaded into the project you're building. The most popular tool among Clojurists for managing artifacts is Leiningen, and this appendix will show you how to use it. You'll also learn how to use Leiningen to totally enhancify your development experience with *plug-ins*.

The Artifact Ecosystem

Because Clojure is hosted on the Java Virtual Machine (JVM), Clojure artifacts are distributed as JAR files (covered in Chapter 12). Java land already has an entire artifact ecosystem for handling JAR files, and Clojure uses it. *Artifact ecosystem* isn't an official programming term; I use

it to refer to the suite of tools, resources, and conventions used to identify and distribute artifacts. Java's ecosystem grew up around the Maven build tool, and because Clojure uses this ecosystem, you'll often see references to Maven. Maven is a huge tool that can perform all kinds of wacky project management tasks. Thankfully, you don't need to get your PhD in Mavenology to be an effective Clojurist. The only feature you need to know is that Maven specifies a pattern for identifying artifacts that Clojure projects adhere to, and it also specifies how to host these artifacts in Maven *repositories*, which are just servers that store artifacts for distribution.

Identification

Maven artifacts need a *group ID*, an *artifact ID*, and a *version*. You can specify these for your project in the *project.clj* file. Here's what the first line of *project.clj* looks like for the `clojure-noob` project you created in Chapter 1:

```
(defproject clojure-noob "0.1.0-SNAPSHOT"
```

`clojure-noob` is both the group ID and the artifact ID of your project, and `"0.1.0-SNAPSHOT"` is its version. In general, versions are permanent; if you deploy an artifact with version 0.1.0 to a repository, you can't make changes to the artifact and deploy it using the same version number. You'll need to change the version number. (Many programmers like the Semantic Versioning system, which you can read about at *http://semver.org/*.) If you want to indicate that the version is a work in progress and you plan to keep updating it, you can append `-SNAPSHOT` to your version number.

If you want your group ID to be different from your artifact ID, you can separate the two with a slash, like so:

```
(defproject group-id/artifact-id "0.1.0-SNAPSHOT"
```

Often, developers will use their company name or their GitHub username as the group ID.

Dependencies

Your *project.clj* file also includes a line that looks like this, which lists your project's dependencies:

```
  :dependencies [[org.clojure/clojure "1.7.0"]]
```

If you want to use a library, add it to this dependency vector using the same naming schema that you use to name your project. For example, if you want to easily work with dates and times, you could add the clj-time library, like this:

```
  :dependencies [[org.clojure/clojure "1.7.0"]
                 [clj-time "0.9.0"]]
```

The next time you start your project, either by running it or by starting a REPL, Leiningen will automatically download clj-time and make it available within your project.

The Clojure community has created a multitude of useful libraries, and a good place to look for them is the Clojure Toolbox at *http://www.clojure-toolbox.com*, which categorizes projects according to their purpose. Nearly every Clojure library provides its identifier at the top of its README, making it easy for you to figure out how to add it to your Leiningen dependencies.

Sometimes you might want to use a Java library, but the identifier isn't as readily available. If you want to add Apache Commons Email, for example, you have to search online until you find a web page that contains something like this:

```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-email</artifactId>
    <version>1.3.3</version>
</dependency>
```

This XML is how Java projects communicate their Maven identifier. To add it your Clojure project, you'd change your `:dependencies` vector so it looks like this:

```
  :dependencies [[org.clojure/clojure "1.7.0"]
                 [clj-time "0.9.0"]
                 [org.apache.commons/commons-email "1.3.3"]]
```

The main Clojure repository is Clojars (*https://clojars.org/*), and the main Java repository is The Central Repository (*http://search.maven.org/*), which is often referred to as just *Central* in the same way that San Francisco residents refer to San Francisco as *the city*. You can use these sites to find libraries and their identifiers.

To deploy your own projects to Clojars, all you have to do is create an account there and run `lein deploy clojars` in your project. This task generates everything necessary for a Maven artifact to be stored in a repository, including a POM file (which I won't go into) and a JAR file. Then it uploads them to Clojars.

Plug-Ins

Leiningen lets you use *plug-ins*, which are libraries that help you when you're writing code. For example, the Eastwood plug-in is a Clojure lint tool; it identifies poorly written code. You'll usually want to specify your plug-ins in the file *$HOME/.lein/profiles.clj*. To add Eastwood, you'd change *profiles.clj* to look like this:

```
{:user {:plugins [[jonase/eastwood "0.2.1"]] }}
```

This enables an `eastwood` Leiningen task for all your projects, which you can run with `lein eastwood` at the project's root.

Leiningen's GitHub project page has excellent documentation on how to use profiles and plug-ins, and it includes a handy list of plug-ins.

Summary

This appendix focused on the aspects of project management that are important but that are difficult to find out about, like what Maven is and Clojure's relationship to it. It showed you how to use Leiningen to name your project, specify dependencies, and deploy to Clojars. Leiningen offers a lot of functionality for software development tasks that don't involve actually writing your code. If you want to find out more, check out the Leiningen tutorial online at *https://github.com/technomancy/leiningen/blob/stable/doc/TUTORIAL.md/*.

B

BOOT, THE FANCY CLOJURE BUILD FRAMEWORK

Boot is an alternative to Leiningen that provides the same functionality. Leiningen's more popular (as of the summer of 2015), but I personally like to work with Boot because it's easier to extend. This appendix explains Boot's underlying concepts and guides you through writing your first Boot tasks. If you're interested in using Boot to build projects right this second, check out its GitHub README (*https://github.com/boot-clj/boot/*) and its wiki (*https://github.com/boot-clj/boot/wiki/*).

NOTE *As of this writing, Boot has limited support for Windows. The Boot team welcomes contributions!*

Boot's Abstractions

Created by Micha Niskin and Alan Dipert, Boot is a fun and powerful addition to the Clojure tooling landscape. On the surface, it's a convenient way to build Clojure applications and run Clojure tasks from the command line. Dig a little deeper and you'll see that Boot is like the Lisped-up lovechild of Git and Unix in that it provides abstractions that make it more pleasant to write code that exists at the intersection of your operating system and your application.

Unix provides abstractions that we're all familiar with to the point where we take them for granted. (Would it kill you to take your computer out to a nice restaurant once in a while?) The process abstraction lets you reason about programs as isolated units of logic that can be easily composed into a stream-processing pipeline through the STDIN and STDOUT file descriptors. These abstractions make certain kinds of operations, like text processing, very straightforward.

Similarly, Boot provides abstractions that make it easy to compose independent operations into the kinds of complex, coordinated operations that build tools end up doing, like converting ClojureScript into JavaScript. Boot's task abstraction lets you easily define units of logic that communicate through *filesets*. The fileset abstraction keeps track of the evolving build context and provides a well-defined, reliable method of task coordination.

That's a lot of high-level description, which hopefully has hooked your attention. But I would be ashamed to leave you with a plateful of metaphors. Oh no, dear reader, that was only the appetizer. Throughout the rest of this appendix, you'll learn how to build your own Boot tasks. Along the way, you'll discover that build tools can actually have a conceptual foundation.

Tasks

Like make, rake, grunt, and other build tools of yore, Boot lets you define tasks. *Tasks* are named operations that take command line options dispatched by some intermediary program (make, rake, Boot).

Boot provides the dispatching program, *boot*, and a Clojure library that makes it easy for you to define named operations and their command line options with the `deftask` macro. To see what all the fuss is about, let's create your first task. Normally, programming tutorials encourage you to write code that prints "Hello World," but I like my examples to have real-world utility, so your task is to print "My pants are on fire!" This information is objectively more useful. First, install Boot; then create a new directory named *boot-walkthrough*, navigate to that directory, create a file named *build.boot,* and write this:

```
(deftask fire
  "Prints 'My pants are on fire!'"
  []
  (println "My pants are on fire!"))
```

Now run this task from the command line with `boot fire`; you should see the message you wrote printed to your terminal. This task demonstrates two out of the three task components: the task is named (`fire`), and it's dispatched by boot. This is super cool. You've essentially created a Clojure shell script, stand-alone Clojure code that you can run from the command line with ease. No *project.clj*, directory structure, or namespaces needed!

Let's extend the example to demonstrate how you'd write command line options:

```
(deftask fire
  "Announces that something is on fire"
  [t thing     THING str  "The thing that's on fire"
   p pluralize       bool "Whether to pluralize"]
  (let [verb (if pluralize "are" "is")]
    (println "My" thing verb "on fire!")))
```

Try running the task like so:

```
boot fire -t heart
# => My heart is on fire!

boot fire -t logs -p
# => My logs are on fire!
```

In the first instance, either you're newly in love or you need to be rushed to the emergency room. In the second, you are a Boy Scout awkwardly expressing your excitement over meeting the requirements for a merit badge. In both instances, you were able to easily specify options for the task.

This refinement of the `fire` task introduced two command line options, `thing` and `pluralize`. Both options are defined using a *domain-specific language (DSL)*. DSLs are their own topic, but briefly, the term refers to mini-languages that you can use within a larger program to write compact, expressive code for narrowly defined domains (like defining options).

In the option `thing`, `t` specifies its short name, and `thing` specifies its long name. `THING` is a bit complicated, and I'll get to it in a second. `str` specifies the option's type, and Boot uses that to validate the argument and convert it. `"The thing that's on fire"` is the documentation for the option. You can view a task's documentation in the terminal with `boot task-name -h`:

```
boot fire -h
# Announces that something is on fire
#
# Options:
#   -h, --help         Print this help info.
#   -t, --thing THING  Set the thing that's on fire to THING.
#   -p, --pluralize    Whether to pluralize
```

Pretty groovy! Boot makes it very easy to write code that's meant to be invoked from the command line.

Now, let's look at `THING`. `THING` is an *optarg*, and it indicates that this option expects an argument. You don't have to include an optarg when you're defining an option (notice that the `pluralize` option has no optarg). The optarg doesn't have to correspond to the full name of the option; you could replace `THING` with `BILLY_JOEL` or whatever you want and the task would work the same. You can also designate complex options using the optarg. (Visit *https://github.com/boot-clj/boot/wiki/Task-Options-DSL#complex-options* for Boot's documentation on the subject.) Basically, complex options allow you to specify that option arguments should be treated as maps, sets, vectors, or even nested collections. It's pretty powerful.

Boot provides you with all the tools you could ask for to build command line interfaces with Clojure. And you've only just started learning about it!

The REPL

Boot comes with a number of useful built-in tasks, including a REPL task. Run `boot repl` to fire up that puppy. The Boot REPL is similar to Leiningen's in that it handles loading your project code so you can play around with it. You might not think this applies to the project you've been writing because you've only written tasks, but you can actually run tasks in the REPL (I've omitted the `boot.user=>` prompt). You can specify options using a string:

```
(fire "-t" "NBA Jam guy")
; My NBA Jam guy is on fire!
; => nil
```

Notice that the option's value comes right after the option.

You can also specify an option using a keyword:

```
(fire :thing "NBA Jam guy")
; My NBA Jam guy is on fire!
; => nil
```

You can also combine options:

```
(fire "-p" "-t" "NBA Jam guys")
; My NBA Jam guys are on fire!
; => nil

(fire :pluralize true :thing "NBA Jam guys")
; My NBA Jam guys are on fire!
; => nil
```

And of course, you can use `deftask` in the REPL as well—it's just Clojure, after all. The takeaway is that Boot lets you interact with your tasks as Clojure functions, because that's what they are.

Composition and Coordination

If what you've seen so far was all that Boot had to offer, it'd be a pretty swell tool, but it wouldn't be very different from other build tools. One feature that sets Boot apart is how it lets you compose tasks. For comparison's sake, here's an example Rake invocation (Rake is the premier Ruby build tool):

```
rake db:create db:migrate db:seed
```

This code will create a database, run migrations on it, and populate it with seed data when run in a Rails project. However, worth noting is that Rake doesn't provide any way for these tasks to communicate with each other. Specifying multiple tasks is just a convenience, saving you from having to run `rake db:create; rake db:migrate; rake db:seed`. If you want to access the result of Task A within Task B, the build tool doesn't help you; you have to manage that coordination yourself. Usually, you'll do this by shoving the result of Task A into a special place on the filesystem and then making sure Task B reads that special place. This looks like programming with mutable, global variables, and it's just as brittle.

Handlers and Middleware

Boot addresses this task communication problem by treating tasks as *middleware factories*. If you're familiar with Ring, Boot's tasks work very similarly, so feel free to skip to "Tasks Are Middleware Factories" on page 287. If you're not familiar with the concept of middleware, allow me to explain! *Middleware* refers to a set of *conventions* that programmers adhere to so they can flexibly create domain-specific function pipelines. That's pretty dense, so let's un-dense it. I'll discuss the *flexible* part in this section and cover *domain-specific* in "Filesets" on page 288.

To understand how the middleware approach differs from run-of-the-mill function composition, here's an example of composing everyday functions:

```
(def strinc (comp str inc))
(strinc 3)
; => "4"
```

There's nothing interesting about this function composition. In fact, this function composition is so unremarkable that it strains my abilities as a writer to actually say anything about it. There are two functions, each does its own thing, and now they've been composed into one. Whoop-dee-doo!

Middleware introduces an extra step to function composition, giving you more flexibility in defining your function pipeline. Suppose, in the

preceding example, that you wanted to return `"I don't like the number X"` for arbitrary numbers but return a string-ified number for everything else. Here's how you could do that:

```
(defn whiney-str
  [rejects]
  {:pre [(set? rejects)]}
  (fn [x]
    (if (rejects x)
      (str "I don't like " x)
      (str x))))

(def whiney-strinc (comp (whiney-str #{2}) inc))
(whiney-strinc 1)
; => "I don't like 2"
```

Now let's take it one step further. What if you want to decide whether or not to call `inc` in the first place? Listing B-1 shows how you could do that:

```
(defn whiney-middleware
  [next-handler rejects]
  {:pre [(set? rejects)]}
  (fn [x]
❶   (if (= x 1)
      "I'm not going to bother doing anything to that"
      (let [y (next-handler x)]
        (if (rejects y)
          (str "I don't like " y)
          (str y))))))

(def whiney-strinc (whiney-middleware inc #{2}))
(whiney-strinc 1)
; => "I don't like 2"
```

Listing B-1: The middleware approach to function composition lets you introduce choice

Here, instead of using `comp` to create your function pipeline, you pass the next function in the pipeline as the first argument to the middleware function. In this case, you're passing `inc` as the first argument to `whiney-middleware` as `next-handler`. `whiney-middleware` then returns an anonymous function that closes over `inc` and has the ability to choose whether to call it or not. You can see this choice at ❶.

We say that a middleware takes a handler as its first argument and returns a handler. In this example, `whiney-middleware` takes a handler as its first argument, `inc`, and it returns another handler, the anonymous function with `x` as its only argument. Middleware can also take extra arguments, like `rejects`, that act as configuration. The result is that the handler returned by the middleware can behave more flexibly (thanks to configuration), and it has more control over the function pipeline (because it can choose whether or not to call the next handler).

Tasks Are Middleware Factories

Boot takes this pattern of making function composition more flexible one step further by separating middleware configuration from handler creation. First, you create a function that takes *n* configuration arguments. This is the *middleware factory*, and it returns a middleware function. The middleware function expects one argument, the next handler, and it returns a handler, just like in the preceding example. Here's a whiney middleware factory:

```
(defn whiney-middleware-factory
  [rejects]
  {:pre [(set? rejects)]}
  (fn [handler]
    (fn [x]
      (if (= x 1)
        "I'm not going to bother doing anything to that"
        (let [y (handler x)]
          (if (rejects y)
            (str "I don't like " y " :'(")
            (str y)))))))

(def whiney-strinc ((whiney-middleware-factory #{3}) inc))
```

As you can see, this code is nearly identical to Listing B-1. The change is that the topmost function, `whiney-middleware-factory`, now only accepts one argument, `rejects`. It returns an anonymous function, the middleware, which expects one argument, a handler. The rest of the code is the same.

In Boot, tasks can act as middleware factories. To show this, let's split the `fire` task into two tasks: `what` and `fire` (see Listing B-2). `what` lets you specify an object and whether it's plural, and `fire` announces that it's on fire. This is great modular software engineering because it allows you to add other tasks, like `gnomes`, to announce that a thing is being overrun with gnomes, which is just as objectively useful. (As an exercise, try creating the gnome task. It should compose with the `what` task, just as `fire` does.)

```
(deftask what
  "Specify a thing"
  [t thing     THING str  "An object"
   p pluralize       bool "Whether to pluralize"]
  (fn middleware [next-handler]
❶   (fn handler [fileset]
      (next-handler (merge fileset {:thing thing :pluralize pluralize})))))

(deftask fire
  "Announce a thing is on fire"
  []
  (fn middleware [next-handler]
❷   (fn handler [fileset]
      (let [verb (if (:pluralize fileset) "are" "is")]
```

```
        (println "My" (:thing fileset) verb "on fire!")
        fileset))))
```

Listing B-2: The full code for composable Boot tasks that announce something's on fire

Here's how you'd run this on the command line:

```
boot what -t "pants" -p - fire
```

And here's how you'd run it in the REPL:

```
(boot (what :thing "pants" :pluralize true) (fire))
```

Wait a minute, what's that `boot` call doing there? And what's with `fileset` at ❶ and ❷? In Micha's words, "The `boot` macro takes care of setup and cleanup (creating the initial fileset, stopping servers started by tasks, things like that). Tasks are functions, so you can call them directly, but if they use the fileset, they will fail unless you call them via the `boot` macro." Let's take a closer look at filesets.

Filesets

Earlier I mentioned that middleware is for creating *domain-specific* function pipelines. All that means is that each handler expects to receive domain-specific data and returns domain-specific data. With Ring, for example, each handler expects to receive a request map representing the HTTP request, which might look something like this:

```
{:server-port 80
 :request-method :get
 :scheme :http}
```

Each handler can choose to modify this request map in some way before passing it on to the next handler, say, by adding a `:params` key with a nice Clojure map of all query string and POST parameters. Ring handlers return a *response map*, which consists of the keys `:status`, `:headers`, and `:body`, and once again each handler can transform this data in some way before returning it to its parent handler.

In Boot, each handler receives and returns a *fileset*. The fileset abstraction lets you treat files on your filesystem as immutable data, and this is a great innovation for build tools because building projects is so file-centric. For example, your project might need to place temporary, intermediary files on the filesystem. Usually, with most build tools, these files get placed in some specially named place, say, *project/target/tmp*. The problem with this is that *project/target/tmp* is effectively a global variable, and other tasks can accidentally muck it up.

Boot's fileset abstraction solves this problem by adding a layer of indirection on top of the filesystem. Let's say Task A creates File X and tells the fileset to store it. Behind the scenes, the fileset stores the file in an anonymous, temporary directory. The fileset then gets passed to Task B, and Task B modifies File X and asks the fileset to store the result. Behind the scenes, a new file, File Y, is created and stored, but File X remains untouched. In Task B, an updated fileset is returned. This is the equivalent of doing `assoc-in` with a map: Task A can still access the original fileset and the files it references.

And you didn't even use any of this cool file management functionality in the `what` and `fire` tasks in Listing B-2! Nevertheless, when Boot composes tasks, it expects handlers to receive and return fileset records. Therefore, to convey your data across tasks, you sneakily added it to the fileset record using `(merge fileset {:thing thing :pluralize pluralize})`.

Although that covers the basic concept of a middleware factory, you'll need to learn a bit more to take full advantage of filesets. The mechanics of working with filesets are all explained in the fileset wiki (*https://github.com/boot-clj/boot/wiki/Filesets*). In the meantime, I hope this information gave you a good conceptual overview!

Next Steps

The point of this appendix was to explain the concepts behind Boot. However, Boot also has a bunch of other functions, like `set-env!` and `task-options!`, that make your programming life easier when you're actually using it. It offers amazing magical features, like providing classpath isolation so you can run multiple projects using one JVM, and letting you add new dependencies to your project without having to restart your REPL. If Boot tickles your fancy, check out its README for more information on real-world usage. Also, its wiki provides top-notch documentation.

FAREWELL!

As Semisonic's hit '90s song "Closing Time" teaches us, every new beginning comes from some other beginning's end.

Congratulations, noble reader, on completing this Clojure journey. I hope you've found it rewarding, and I hope you're looking forward to more!

And believe me, there's so much more. One of the things I like most about Clojure is that there's an entire world to explore. Logic programming, parsers, type systems—name a fun realm of computer science, and you can investigate it with Clojure. I leave you with my humble suggestions of where to go next.

If you want to get a broad overview of the Clojure ecosystem, check out *http://www.clojure-toolbox.com/*, which lists hundreds of Clojure projects organized by the problem they solve.

If you're interested in web programming, a great place to start is the Luminus framework (*http://www.luminusweb.net/*). The documentation is excellent, and you'll have a website running in no time.

To stay up-to-date with Clojure news, a great resource is Eric Normand's Clojure Gazzette (*http://www.clojuregazette.com/*). There's also the Clojure mailing list, of course (*https://groups.google.com/forum/#!forum/clojure*) and the Clojure subreddit is a helpful, friendly place, too (*http://www.reddit.com/r/clojure*).

If Twitter is your social media outlet of choice, then @swannodette (David Nolen), @gigasquid (Carin Meier), @puredanger (Alex Miller), @ztellman (Zach Tellman), @bbatsov (Bozhidar Batsov), and @stuartsierra (Stuart Sierra) are your huckleberries. You could also follow me, @nonrecursive!

Farewell, little teacup, and have fun Clojuring!

INDEX

Symbols

+ (addition operator), 36–37
@, 155, 197
>!! (blocking put), 235–238
<!! (blocking take), 235–238
. (dot operator), 258–259
= (equality operator), 39
#', 128
>! (parking put), 235–238
<! (parking take), 235–238
' (quote) reader macro, 154–155
& (rest parameter), 54

A

abstractions, 265–266
    implementing, 270
    through indirection, 77
    with macros, 183
abstract syntax tree (AST), 149
add-watch, 216
alias, 132
alter, 219–221
alter-var-root function, 227
and (Boolean operator), 40
    source code, 168
apply function, 91
architecture, of code, 110
arity, 52–54
    overloading, 52–53
artifact ecosystem, 277–280
artifacts, 277
assoc-in, 114
AST (abstract syntax tree), 149
asynchronous tasks, 191
atomic values, 210
atoms, 212–217
auto-gensym, 177–178

B

binding
    with def, 40–41
    dynamic vars, 223–227
    with let, 61–63
    local, 157–158
blocking, 191
blocking put (>!!), 235–238
blocking take (<!!), 235–238
Boolean
    expressions, 39–40
    forms, 37–38
    operators, 40
    values, 39
Boot
    classpath isolation, 289
    composition, 285–288
    deftask, 282
    documentation, 283
    filesets, 282, 288–289
    middleware, 285–286
    middleware factories, 285, 287–288
    optarg, 284
    REPL, 284
    tasks, 282–284
bound-fn, 227
Brave and True Ale example, 180

C

callbacks
    hell, 244
    with promises, 202
Central Repository, 279
chan, 235
channels, 235–237
    buffering, 236–237
    timeout, 242

`char`, 120
cheese heist, 130, 140–144
CIDER package, 23–28
    handling errors, 27–28
    installation, 23
    key bindings, 25–27
    starting, 23
class instantiation, 272
classpath, 252
Clojars repository, 279
Clojure
    compiler, 4
    hosted language, 4
    metaphysics, 210–211
clojure.jar, 255–257
closures, 58
collection abstraction, 88–90
command-line interaction, 121–124
comma-separated values (CSV), 93
`commute`, 221–223
compare-and-set, 214–215
`comp` function, 105–106, 120
compilation, 248–252
`complement` function, 92–93
`concat` function, 84
concurrency, 190–196
    dining philosophers problem, 195–196
    dwarven berserker problem, 195
    mutual exclusion problem, 194–195, 210
        preventing with delays, 199
    nondeterministic execution, 208–210
    queues, 243–244
    reference cell problem, 193–194, 207, 209
        preventing with promises, 202
    stateless, 228–232
    tasks, 190
    The Three Concurrency Goblins, 193–196
`conj` function, 45, 46, 47, 90
`cons` function, 74–77
`contains?` function, 47
control flow, 37–40
    Boolean expressions, 39–40
    `do` operator, 38
    `if` expression, 37–38
    `when` operator, 38–39
core.async
    `alts!`, 235, 242
    `alts!!`, 235, 241–243
    blocking, 237–238
    buffering, 236–237
    events, 233
    hot dog vending machine example, 239–241
    parking, 237–238
    pipelines, 240, 244–245
    put, 235–238
    queues, 243–244
    take, 235–238
    thread, 238–239
    timeout channels, 235, 242
    waiting, 235–238
`create-ns`, 129
CSV (comma-separated values), 93
cuddle zombie, 208

D

data structures, 41–48
    immutable, 42, 100–105, 210
    keywords, 44–45
    lists, 45–46
    maps, 43–44
    numbers, 42
    sets, 46–47
    simplicity, 48
    strings, 42–43
    vectors, 45
data types, 265–266
    extending, 270
    instances, 266
`def`
    naming values, 40–41
    storing objects, 127–129
`defprotocol`, 270
`defrecord`, 272
`deftype`, 275
delays, 196, 198–199

deliver, 200
dependencies, 277, 278–279
deref, 128, 197
    reader macro, 155
    timeout, 202
dereferencing
    atoms, 212
    delays, 198–199
    futures, 191–198
    promises, 200–202
destructuring, 54–56
dispatching function, 266, 267–268
dispatching value, 266, 267–268
dispatch value, 267–268
distributed computing, 191
domain-specific language (DSL), 283
do operator, 38
dorun function, 229
doseq, 121
dosync, 219–221
doto macro, 260
dot special form, 258
dot operator (.), 258–259
drop function, 81
drop-while function, 81–82
DSL (domain-specific language), 283
Dungeons and Dragons, 265
dynamic binding, 223–227

E

Eastwood plug-in (lint tool), 279–280
editors, 9
El Chupacabra, 218
elisp, 11, 17, 19, 72
Emacs
    buffers, 14–15
    CIDER package, 23. *See also* CIDER package
    configuration, 13
    cursor, 20
    customizing, 15–16
    files, 15–17
    frames, 24
    help, 22
    installation, 12–14
    key bindings, 17
    killing, 21–22
    kill ring, 21–22
    Lisp (elisp), 11, 19
    mark, 20
    minibuffer, 15
    modes, 18–19
        line, 18
        major, 18
        minor, 19
    movement, 20
    packages, 19
    Paredit, 28–30. *See also* Paredit
    point, 20
    regions, 20
    windows, 24
    yank, 21
empty? function, 88
equality operator (=), 39
eval, 151
evaluation, 36
    lists, 159–160
    macros, 160–162
    model, 148–152
    rules, 155–162
    to self, 156
    symbols, 156
evaluator, 148, 149–152, 155
expression, 36
    Boolean, 39–40
    function, 98
    if, 37–38
extend-protocol, 271, 274
extend-type, 270, 274

F

false (value), 39
falsey values, 39
fields, 272
file naming conventions, 134
filter function, 83
first function, 74
force, 198
forms, 36–37
fully qualified symbols, 171

functional programming, 79, 97
immutable data structures, 100–105
Peg Thing game, 108–124
pure functions, 98–100
functions, 48–59
anonymous, 57–58
arity, 52–54
overloading, 52–53
calls, 48–51, 159
composition, 103–105, 285
defining, 51–52
expression, 48
higher-order, 49
pure, 98–100, 105–107
rest parameters, 54
futures, 196–198, 202–205

G

gensym, 177–178
`get` function, 43–44, 45, 47
`get-in` function, 44
go blocks, 235–239
grain size, 230

H

Handy, Jack, 19
`hash-map` function, 43
`hash-set`, 46
head (of a sequence), 65
Hickey, Rich, 4
hierarchical dispatching, 269
hobbits
modeling, 59–61
targeting, 67–68
homoiconic languages, 148, 152
hot dog vending machine, 239–243
humans, 265

I

identity (Clojure metaphysics), 211
`identity` function, 95
`if` expressions, 37–38
`if-let`, 119
immutable data structures, 100–105, 210, 212
implementing abstractions, 74
importing
Java classes, 253–254
record types, 273
`in-ns` function, 129
installation
Emacs, 12–14
CIDER package, 23
packages, 19
Leiningen, 5
instance, of a data type, 266
interfaces, 77
interleaving, 190, 192–193
interning, 128
`into` function, 88–89

J

JAR files, 4, 249, 255
Java
bytecode, 4, 248–249, 252
classes, 273
classpath, 252, 254–255
entry point, 255
imports, 253–255
interop. *See* Java interop
JAR files, 4, 249, 255
`main` method, 252, 255, 257
packages, 253–255
stacks, 259–260
`javac`, 252
Java interop, 250, 257–261
creating objects, 259–260
`Date` class, 262
files, 262–264
importing, 260–261
input/output, 262–264
method calls, 258
mutating objects, 259–260
passing arguments, 258
syntax, 258–259
`System` class, 261–262
Java Virtual Machine (JVM), 4, 150, 248–249
threads, 191–193
just-in-time compilation, 248

K

key functions, 84
keywords, 44–45

L

Lady Gaga, 190–191
lazy sequences, 84–88, 112
    chunking, 86
    defining, 87
    efficiency, 84–87
    infinite, 87–88
    realizing, 84
    `repeatedly` function, 87
    `repeat` function, 87
Leiningen build tool, 5–8, 277–280
    dependencies, 278–279
    identification, 278
    plug-ins, 279–280
`let`, 61–63
`line-seq` function, 264
linked list, 74–76
lint tool (Eastwood plug-in), 279–280
Lisp, 4, 11, 36, 150
`list` function, 46
lists, 45–46
    evaluation rules for, 159–160
literals, 36
local binding, 157–158
`loop`, 63–64

M

`macroexpand`, 162
macros, 147–185
    argument destructuring, 167–168
    Brave and True Ale example, 180–184
    building lists for evaluation, 168–173
    calling, 50–51
    characters, 154
    defining, 167
    distinguishing symbols and values, 168–169
    evaluation rules for, 160–162
    expansion, 162
    gotchas, 176–180
        composition, 179–180
        double evaluation, 178
        variable capture, 176–178
    infection, 166
`map` function, 50, 73, 79–80
maps (data structure), 43–44
    destructuring, 55
Maven, 278, 279
McCarthy, John, 148
McFishwich, 85
`memoize` function, 107
metaphysics, Clojure, 210–211
multimethods, 266–269
    default, 268

N

names
    in Clojure metaphysics, 211
    collision, 128–129
    for values, 40–41
namespaces, 126
    aliasing, 136
    `create-ns`, 129
    creating and switching to, 129–130
    current, 126
    `in-ns`, 129
    `ns` macro, 126, 129, 138–140
    `ns-interns`, 128
    `ns-map`, 128
    `refer-clojure`, 138
    reference, 138
    refering, 130–132, 135
    requiring, 134–137
    `user`, 126
    using, 136–137
`nil` (value), 39, 47
nondeterministic execution, 193
`not-empty`, 120
`ns` macro, 126, 129, 138–140
`ns-interns`, 128
`ns-map`, 128
`nth` function, 46
numbers, 42

O

object-oriented metaphysics, 208–210
object-oriented programming, 104, 250–251
  classes, 251
  methods, 251
  objects, 250–251
operators, 36
  addition (+), 36–37
  Boolean
    `and`, 40
    `or`, 40
  `do`, 38
  dot (`.`), 258–259
  equality (`=`), 39
  `when`, 38–39
`or` (Boolean operator), 40

P

parallelism, 190–196. *See also* concurrency
Paredit, 28
  barfing, 29–30
  navigation, 30
  slurping, 29
  wrapping, 29
`partial` function, 91–92, 120
Peg Thing game, 108–124
Perlis, Alan, 48
philosophy (of Clojure), 48
plug-ins, 279–280
`pmap`, 228–232
polymorphism, 77, 266
  multimethods, 266–269
  protocols, 77, 269–272
predicate functions, 81–82, 119–120
processes, 234–239
  blocking, 237–238
  buffering, 236–237
  parking, 237–238
  `thread`, 238–239
programming to abstractions, 72
  indirection, 77–78
  linked lists, 74–77
  sequences abstraction, 72–74
projects
  building, 7
  creating, 5–6
  organizing, 133–140
  running, 6
promises, 196, 200–205
protocols, 77, 269–272
`proxy`, 275
pure functions, 98–100, 117

Q

queues
  macro, 202–205
  processes, 243–244
`quote`, 160
quote (`'`) reader macro, 154–155
quoting, 127, 169
  simple, 169–171
  syntax, 171–174
  unquote splicing, 174–176
  unquoting, 172, 175
  with `when`, 170

R

reader, 148, 150, 153–155, 264
  form, 128, 153
  macros, 57, 153, 154–155
`read-string` function, 153–154
`realized?`, 198
records, 272–274
`recur`, 102–103
recursion, 100–103
`reduce` function, 66–67, 80–81, 114
reducers library, 231–232
`refer`, 130–132
reference types, 211
  atoms, 212–215
referential transparency, 98–99
refs, 218–223
regular expressions, 64
`reify`, 275
`repeatedly` function, 87, 228
`repeat` function, 87
REPL, 7–9, 23–24
  Boot, 284
repositories, 278

`require`, 111, 134–137
`rest` function, 74–77

S

scope, 61
Semantic Versioning system, 278
sequence (seq), 73–74
    abstraction, 72–77
    function examples, 79–84
    lazy, 84–88
sets, 46–47
s-expressions, 150
side effects, of functions, 98, 99–100
Simmons, Richard, 165
    were-Simmons, 267
simplicity, of data structures, 48
`slurp` function, 29, 94, 263
sock gnomes, 218–220
software transactional memory (STM), 218
`some` function, 83
`sort-by` function, 84
`sort` function, 84
special forms, 50–51, 156, 159–160
`spit` function, 144, 263
Stallman, Richard, 13
state, 207, 211, 212
    mutable, 208–210
STM (software transactional memory), 218
strings, 42–43
    concatenating, 43
    pattern matching, 64
SVG, 140
`swap!`, 212–215
Swift, Taylor, 9
symbols, 126, 156–158
    fully qualified, 129–130
    resolving, 156–157
synchronous tasks, 191
syntactic abstraction, 162, 163
syntax, 36
    Java interop, 258–259

T

tail call optimization, 102
tail (of a sequence), 65
`take` function, 81
`take-while` function, 81–82
telepath, 225
`thread-bound?` function, 226
threads, 191–193
    delays, 198–199
    futures, 196–198
    nondeterministic programs, 193
    promises, 200–202
    spawning, 192
`Thread/sleep`, 196–197
Tick, The, 53
transactions, 218–221
troll, 225
`true` (value), 39
truthy values, 39
tuples, 119
types, 266

U

`unless` macro, 171
unquote splicing, 174–176
unquoting, 172, 175
`use` function, 136–137

V

validators, 217
values, 210–211
vampire
    data analysis, 93–96
    food journal, 79–84
Vampire Diaries, The, 71
variable assignment, 101
vars, 126–129, 223–227
    binding conveyance, 227
    dynamic binding, 223–226
    interning, 128
    per-thread binding, 226–227
    private, 132
    reader form, 128
    roots, 227

`vector` function, 45
vectors, 45

W

watches, 215–216
were-Simmons, 267
`when` operator, 38–39, 166, 170
whitespace (to separate operands), 36
`with-open`, 264
`with-redefs`, 227

Y

yak butter, 200–202

RESOURCES

Visit *https://www.nostarch.com/clojure/* for resources, errata, and more information.

More no-nonsense books from **NO STARCH PRESS**

LAND OF LISP
Learn to Program in Lisp, One Game at a Time!
by CONRAD BARSKI, M.D.
OCTOBER 2010, 504 PP., $49.95
ISBN 978-1-59327-281-4

LEARN YOU A HASKELL FOR GREAT GOOD!
A Beginner's Guide
by MIRAN LIPOVAČA
APRIL 2011, 400 PP., $44.95
ISBN 978-1-59327-283-8

LEARN YOU SOME ERLANG FOR GREAT GOOD!
A Beginner's Guide
by FRED HÉBERT
JANUARY 2013, 624 PP., $49.95
ISBN 978-1-59327-435-1

ELOQUENT JAVASCRIPT, 2ND EDITION
A Modern Introduction to Programming
by MARIJN HAVERBEKE
DECEMBER 2014, 472 PP., $39.95
ISBN 978-1-59327-584-6

AUTOMATE THE BORING STUFF WITH PYTHON
Practical Programming for Total Beginners
by AL SWEIGART
APRIL 2015, 504 PP., $29.95
ISBN 978-1-59327-599-0

THE ART OF R PROGRAMMING
A Tour of Statistical Software Design
by NORMAN MATLOFF
OCTOBER 2011, 400 PP., $39.95
ISBN 978-1-59327-384-2

PHONE:
800.420.7240 OR
415.863.9900

EMAIL:
SALES@NOSTARCH.COM

WEB:
WWW.NOSTARCH.COM

SHELVE IN: PROGRAMMING LANGUAGES/CLOJURE

JOIN THE RANKS OF NOBLE CLOJURISTS

For weeks, months—nay!—from the very moment you were born, you've felt it calling to you. At long last you'll be united with the programming language you've been longing for: Clojure!

As a Lisp-style functional programming language, Clojure lets you write robust and elegant code, and because it runs on the Java Virtual Machine, you can take advantage of the vast Java ecosystem. ***Clojure for the Brave and True*** offers a "dessert-first" approach: you'll start playing with real programs immediately, as you steadily acclimate to the abstract but powerful features of Lisp and functional programming. Inside you'll find an offbeat, practical guide to Clojure, filled with quirky sample programs that catch cheese thieves and track glittery vampires.

LEARN HOW TO

- Wield Clojure's core functions
- Use Emacs for Clojure development
- Write macros to modify Clojure itself
- Use Clojure's tools to simplify concurrency and parallel programming

Clojure for the Brave and True assumes no prior experience with Clojure, the Java Virtual Machine, or functional programming. Are you ready, brave reader, to meet your true destiny? Grab your best pair of parentheses—you're about to embark on an epic journey into the world of Clojure!

ABOUT THE AUTHOR

Daniel Higginbotham has been a professional programmer for 11 years, half of that at McKinsey & Company, where he used Clojure to build mobile and web applications. He has also contributed to the curriculum for ClojureBridge, an organization that offers free, beginner-friendly Clojure workshops for women. Daniel blogs about life and programming at *http://flyingmachinestudios.com/*, and can be found on Twitter, @nonrecursive. He lives in Durham, North Carolina, with his wife and four cats.

COVERS CLOJURE 1.7
REQUIRES JAVA 1.6 OR LATER

no starch press

www.nostarch.com
THE FINEST IN GEEK ENTERTAINMENT™

ISBN: 978-1-59327-591-4
9 781593 275914
53495
$34.95 ($40.95 CDN)
6 89145 75919 8