

Extracted from:

Programming Clojure, Third Edition

This PDF file contains pages extracted from *Programming Clojure, Third Edition*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2018 The Pragmatic Programmers, LLC.

All rights reserved.

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

The Pragmatic Bookshelf

Raleigh, North Carolina

The
Pragmatic
Programmers

Programming Clojure

Third Edition



Alex Miller
with Stuart Halloway
and Aaron Bedra
edited by Jacquelyn Carter

Programming Clojure, Third Edition

Alex Miller
with Stuart Halloway
and Aaron Bedra

The Pragmatic Bookshelf

Raleigh, North Carolina



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at <https://pragprog.com>.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Copy Editor: Paula Robertson
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact support@pragprog.com.

For international rights, please contact rights@pragprog.com.

Copyright © 2018 The Pragmatic Programmers, LLC.
All rights reserved.

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

Printed in the United States of America.
ISBN-13: 978-1-68050-246-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—February 2018

Introduction

Clojure is a dynamic programming language for the Java Virtual Machine (JVM) with a compelling combination of features:

- *Clojure is elegant.* Clojure's clean, careful design lets you write programs that get right to the essence of a problem, without a lot of clutter and ceremony.
- *Clojure is Lisp reloaded.* Clojure has the power inherent in Lisp but is not constrained by the history of Lisp.
- *Clojure is a functional language.* Data structures are immutable, and most functions are free from side effects. This makes it easier to write correct programs and to compose large programs from smaller ones.
- *Clojure simplifies concurrent programming.* Many languages build a concurrency model around locking, which is difficult to use correctly. Clojure provides several alternatives to locking: software transactional memory, agents, atoms, and dynamic variables.
- *Clojure embraces Java.* Calling from Clojure to Java is direct and fast, with no translation layer.
- *Unlike many popular dynamic languages, Clojure is fast.* Clojure is written to take advantage of the optimizations possible on modern JVMs.

Many other languages cover *some* of the features described in the previous list. Of all these languages, Clojure stands out. The individual features listed earlier are powerful and interesting. Their clean synergy in Clojure is compelling. We will cover all these features and more in [Chapter 1, Getting Started, on page ?](#).

Who This Book Is For

Clojure is a powerful, general-purpose programming language. As such, this book is for programmers with experience in a programming language like

Java, C#, Python, or Ruby, but new to Clojure and looking for a powerful, elegant language.

Clojure is built on top of the Java Virtual Machine, and it is *fast*. This book will be of particular interest to Java programmers who want the expressiveness of a dynamic language without compromising on performance.

Clojure is helping to redefine what features belong in a general-purpose language. If you program in Lisp, use a functional language such as Haskell, or write explicitly concurrent programs, you'll enjoy Clojure. Clojure combines ideas from Lisp, functional programming, and concurrent programming and makes them more approachable to programmers seeing these ideas for the first time.

Clojure is part of a larger phenomenon. Languages such as Erlang, F#, Haskell, and Scala have garnered attention recently for their support of functional programming or their concurrency model. Enthusiasts of these languages will find much common ground with Clojure.

What's in This Book

[Chapter 1, *Getting Started*, on page ?](#) demonstrates Clojure's elegance as a general-purpose language, plus the functional style and concurrency model that make Clojure unique. It also walks you through installing Clojure and developing code interactively at the REPL.

[Chapter 2, *Exploring Clojure*, on page ?](#) is a breadth-first overview of all of Clojure's core constructs. After this chapter, you'll be able to read most day-to-day Clojure code.

The next two chapters cover functional programming. [Chapter 3, *Unifying Data with Sequences*, on page ?](#) shows how all data can be unified under the powerful *sequence* metaphor.

[Chapter 4, *Functional Programming*, on page ?](#) shows you how to write functional code in the same style used by the sequence library.

[Chapter 5, *Specifications*, on page ?](#) demonstrates how to write specifications for your data structures and functions and use them to aid in development and testing.

[Chapter 6, *State and Concurrency*, on page ?](#) delves into Clojure's concurrency model. Clojure provides four powerful models for dealing with concurrency, plus all of the goodness of Java's concurrency libraries.

[Chapter 7, *Protocols and Datatypes*, on page ?](#) walks through records, types, and protocols in Clojure. These concepts were introduced in Clojure 1.2.0 and enhanced in 1.3.0.

[Chapter 8, *Macros*, on page ?](#) shows off Lisp’s signature feature. Macros take advantage of the fact that Clojure code is data to provide metaprogramming abilities that are difficult or impossible in anything but a Lisp.

[Chapter 9, *Multimethods*, on page ?](#) covers one of Clojure’s answers to polymorphism. Polymorphism usually means “take the *class* of the *first* argument and dispatch a method based on that.” Clojure’s multimethods let you choose *any function* of *all* the arguments and dispatch based on that.

[Chapter 10, *Java Interop*, on page ?](#) shows you how to call Java from Clojure and call Clojure from Java. You’ll see how to take Clojure straight to the metal and get Java-level performance.

Finally, [Chapter 11, *Building an Application*, on page ?](#) provides a view into a complete Clojure workflow. You will build an application from scratch, working through solving the various parts to a problem and thinking about simplicity and quality.

How to Read This Book

All readers should begin by reading the first two chapters in order. Pay particular attention to [Simplicity and Power in Action, on page ?](#), which provides an overview of Clojure’s advantages.

Experiment continuously. Clojure provides an interactive environment where you can get immediate feedback; see [Using the REPL, on page ?](#) for more information.

After you read the first two chapters, skip around as you like. But read [Chapter 3, *Unifying Data with Sequences*, on page ?](#) before you read [Chapter 6, *State and Concurrency*, on page ?](#). These chapters lead you from Clojure’s immutable data structures to a powerful model for writing correct concurrency programs.

As you make the move to longer code examples in later chapters, make sure you use an editor that provides Clojure indentation for you. If you can, try to use an editor that supports parentheses balancing, such as Emacs’ `paredit` mode or the Cursive plugin for IntelliJ. This feature will be a huge help as you’re learning to program in Clojure.

For Functional Programmers

- Clojure's approach to FP strikes a balance between academic purity and the realities of execution on the current generation of JVMs. Read [Chapter 4, *Functional Programming*, on page ?](#) carefully to understand how Clojure idioms differ from languages such as Haskell.
- The concurrency model of Clojure ([Chapter 6, *State and Concurrency*, on page ?](#)) provides several explicit ways to deal with side effects and state and will make FP appealing to a broader audience.

For Java/C# Programmers

- Read [Chapter 2, *Exploring Clojure*, on page ?](#) carefully. Clojure has very little syntax (compared to Java or C#), and we cover the ground rules fairly quickly.
- Pay close attention to macros in [Chapter 8, *Macros*, on page ?](#). These are the most alien part of Clojure when viewed from a Java or C# perspective.

For Lisp Programmers

- Some of [Chapter 2, *Exploring Clojure*, on page ?](#) will be review, but read it anyway. Clojure preserves the key features of Lisp, but it breaks with Lisp tradition in several places, and they are covered here.
- Pay close attention to the lazy sequences in [Chapter 4, *Functional Programming*, on page ?](#).
- If you like Emacs, get a mode for Clojure that makes you happy before working through the code examples in later chapters. There are many options, but consider inf-clojure for a minimalist setup and CIDER for a full-feature environment.

For Perl/Python/Ruby Programmers

- Read [Chapter 6, *State and Concurrency*, on page ?](#) carefully. Intraprocess concurrency is very important in Clojure.
- Embrace macros ([Chapter 8, *Macros*, on page ?](#)). But do not expect to easily translate metaprogramming idioms from your language into macros. Remember always that macros execute at read time, not runtime.

Notation Conventions

The following notation conventions are used throughout the book.

Literal code examples use the following font:

```
(+ 2 2)
```

The result of executing a code example is preceded by `->`.

```
(+ 2 2)
-> 4
```

Where console output cannot easily be distinguished from code and results, it's preceded by a pipe character (`|`).

```
(println "hello")
| hello
-> nil
```

When introducing a Clojure form for the first time, we'll show the grammar for the form like this:

```
(example-fn required-arg)
(example-fn optional-arg?)
(example-fn zero-or-more-arg*)
(example-fn one-or-more-arg+)
(example-fn & collection-of-variable-args)
```

The grammar is informal, using `?`, `*`, `+`, and `&` to document different argument-passing styles, as shown previously.

Clojure code is organized into *libs* (libraries). Where examples in the book depend on a library that's not part of the Clojure core, we document that dependency with a `require` form:

```
(require '[lib-name :refer [var-names+] :as alias])
```

The `require` form has several options. The `:refer` option can be used to make either specific vars (or all vars with `:all`) available in the current namespace. The `:alias` option can be used to create an alias for references to the library. For example, a commonly used function is `file`, from the `clojure.java.io` library:

```
(require '[clojure.java.io :as io])
(io/file "hello.txt")
-> #<File hello.txt>
```

Clojure returns `nil` from a successful call to `require`. For brevity, this is omitted from the example listings.

While reading the book, you'll enter code in an interactive environment called the REPL. The REPL prompt looks like this:

```
user=>
```

The user in the prompt indicates the namespace you're currently working in. For most of the examples, the current namespace is irrelevant. Where the namespace is irrelevant, we use the following syntax for interaction with the REPL:

```
(+ 2 2)      ; input line without namespace prompt
-> 4         ; return value
```

In those instances where the current namespace is important, we use this:

```
user=> (+ 2 2) ; input line with namespace prompt
-> 4         ; return value
```

Web Resources and Feedback

Programming Clojure's official home on the web is the *Programming Clojure* home page¹ at the Pragmatic Bookshelf website. From there, you can order electronic or paper copies of the book and download sample code. You can also offer feedback by submitting errata entries² or posting in the forum.³

Downloading Sample Code

The sample code for the book is available from the following location:

- The *Programming Clojure* home page⁴ links to the official copy of the source code and is updated to match each release of the book.

Individual examples are in the examples directory, unless otherwise noted.

Throughout the book, listings begin with their filename, set apart from the actual code by a gray background. For example, the following listing comes from `src/examples/preface.clj`:

```
src/examples/preface.clj
(println "hello")
```

If you're reading the book in PDF form, you can click the little gray box preceding a code listing to download that listing directly. With the sample code in hand, you are ready to get started. We'll begin by meeting the combination of features that make Clojure unique.

1. <https://www.pragprog.com/titles/shcloj3/programming-clojure>
2. <https://www.pragprog.com/titles/shcloj3/errata>
3. <http://forums.pragprog.com/forums/439>
4. <http://www.pragprog.com/titles/shcloj3>