Extracted from:

Programming Clojure

Second Edition

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Programming Clojure

Second Edition

Stuart Halloway
Aaron Bedra

Foreword by Rich Hickey, creator of Clojure

Edited by Michael Swaine

The team that produced this book includes:

Michael Swaine (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-934356-86-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—April 2012

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 experienced programmers looking for power and elegance. This book will be useful for anyone with experience in a modern programming language such as C#, Java, Python, or Ruby.

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 will 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 Is 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 will 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, *State*, 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 6, *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 7, *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 8, *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 9, *Java Down and Dirty*, on page ? shows you how to call Java from Clojure and call Clojure from Java. You will see how to take Clojure straight to the metal and get Java-level performance.

Finally, Chapter 10, *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. You will use a set of helpful Clojure libraries to produce and deploy a web application.

Appendix 1, *Editor Support*, on page ? lists editor support options for Clojure, with links to setup instructions for each.

How to Read This Book

All readers should begin by reading the first two chapters in order. Pay particular attention to Section 1.1, *Why Clojure?*, 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 5, *State*, 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 the later chapters, make sure you use an editor that provides Clojure indentation for you. Appendix 1, *Editor Support*, on page ? will point you to common editor options. If you can, try to use an editor that supports parentheses balancing, such as Emacs' paredit mode or the CounterClockWise plug-in for eclipse. This feature will be a huge help as you are 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 5, *State*, 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 7, *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 ?.
- Get an Emacs mode for Clojure that makes you happy before working through the code examples in later chapters.

For Perl/Python/Ruby Programmers

- Read Chapter 5, *State*, on page ? carefully. Intraprocess concurrency is very important in Clojure.
- Embrace macros (Chapter 7, *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 is preceded by a pipe character (|).

```
(println "hello")
```

```
| hello
-> nil
```

When introducing a Clojure form for the first time, we will 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 is not part of the Clojure core, we document that dependency with a use or require form:

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

This form of use brings in only the names in var-names, while require creates an alias, making each function's origin clear. For example, a commonly used function is file, from the clojure.java.io library:

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

or the require-based counterpart:

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

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

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

```
user=>
```

The user before the prompt tells the namespace you are currently working in. For most of the book's examples, the current namespace is irrelevant. Where the namespace is irrelevant, we will use the following syntax for interaction with the REPL:

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

In those few instances where the current namespace is important, we will 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 page[1] 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 entries[2] or posting in the forum[3] for the book.

Downloading Sample Code

The sample code for the book is available from one of two locations:

- The *Programming Clojure* home page[4] links to the official copy of the source code and is updated to match each release of the book.
- The *Programming Clojure* git repository[5] is updated in real time. This is the latest, greatest code and may sometimes be *ahead* of the prose in 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 are reading the book in PDF form, you can click the little gray box preceding a code listing and download that listing directly.

With the sample code in hand, you are ready to get started. We will begin by meeting the combination of features that make Clojure unique.

1. http://www.pragprog.com/titles/shcloj2/programming-clojure
2. http://www.pragprog.com/titles/shcloj2/errata
3. http://forums.pragprog.com/forums/207
4. http://www.pragprog.com/titles/shcloj2
5. http://github.com/stuarthalloway/programming-clojure