The Pragmatic Programmers

Programming Clojure

Stuart Halloway

Edited by Susannah Davidson Pfalzer

Prepared exclusively for Michael Kevin Hagel

The book you're reading is still under development. As part of our Beta book program, we're releasing this copy well before we normally would. That way you'll be able to get this content a couple of months before it's available in finished form, and we'll get feedback to make the book even better. The idea is that everyone wins!

Be warned. The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos and other weirdness. And there's been no effort spent doing layout, so you'll find bad page breaks, over-long lines with little black rectangles, incorrect hyphenations, and all the other ugly things that you wouldn't expect to see in a finished book. We can't be held liable if you use this book to try to create a spiffy application and you somehow end up with a strangely shaped farm implement instead. Despite all this, we think you'll enjoy it!

Throughout this process you'll be able to download updated PDFs from your account on http://pragprog.com. When the book is finally ready, you'll get the final version (and subsequent updates) from the same address. In the meantime, we'd appreciate you sending us your feedback on this book at http://pragprog.com/titles/shcloj/errata, or by using the links at the bottom of each page.

Thank you for being part of the Pragmatic community!

► **Andy & Dave**

Prepared exclusively for Michael Kevin Hagel

Programming Clojure

Stuart Halloway

The Pragmatic Bookshelf
Raleigh, North Carolina Dallas, Texas

Prepared exclusively for Michael Kevin Hagel

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 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 courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragprog.com

Copyright © 2008 Stuart Halloway.

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-10: 1-934356-33-6

ISBN-13: 978-1-934356-33-3

Printed on acid-free paper.

B1.0 printing, November 4, 2008

Version: 2008-11-4

Prepared exclusively for Michael Kevin Hagel

Contents

Prepared exclusively for Michael Kevin Hagel

Prepared exclusively for Michael Kevin Hagel

Acknowledgments

Many people have contributed to what is good in this book. The problems and errors that remain are mine alone.

Thanks to my co-workers at Relevance, for creating an atmosphere where good ideas can grow and thrive. Clojure helps to answer questions that working at Relevance has taught me to ask.

Thanks to Jay Zimmerman, and all the speakers and attendees on the No Fluff, Just Stuff conference tour. I have sharpened my ideas about Clojure in conversations with you all over the United States: sometimes in the formal sessions, but equally often in the hotel bar.

Thanks to the kind folks on the Clojure mailing list[1] for all their help and encouragement. Meikel Brandmeyer, Bill Clementson, Brian Doyle, Steve Gilardi, Rich Hickey, Mark Hoemmen, Chris Houser, Achim Passen, Stuart Sierra, Paul Stadig, helped with specific questions I had along the way.

Thanks to everyone at the Pragmatic Bookshelf. Thanks especially to my editor, Susannah Pfalzer, for good advice delivered on a very aggressive schedule. Thanks to Dave Thomas and Andy Hunt for creating a fun platform for writing technical books, and for betting on the passions of their authors.

Thanks to my many technical reviewers for all your comments.

Thanks to Rich Hickey for creating the excellent Clojure language and fostering a community around it.

Finally, thanks to my wife Joey, and my daughters Hattie, Harper, and Mabel Faire. You all make the sun rise.

1. http://groups.google.com/group/clojure

Prepared exclusively for Michael Kevin Hagel

Preface

Clojure is a dynamic programming language for the Java Virtual Machine, 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 functions tend to be side-effect free. This makes it easier to write correct programs, and to compose large programs from smaller ones.
- *Clojure is concurrent.* Of course, Java itself has pretty good concurrency support. But, there is wide agreement that lock-based concurrency is difficult to use correctly. Clojure provides alternatives to lock-based concurrency: software transactional memory, agent, and dynamic variables.
- *Clojure embraces Java.* Calling from Clojure to Java is direct, and goes through no translation layer.
- Unlike many popular dynamic languages, *Clojure is fast*. Wherever you need it, you can get the exact same performance that you could get from hand-written Java code.

Many other languages cover *some* of the features described above. My personal quest for Clojure included significant time spent with Ruby, Python, and JavaScript, plus less intensive exploration of Scala, Groovy, and Fan. These are all good languages, and they all simplify writing code on the Java platform.

But for me, Clojure stands out. The individual features above are powerful and interesting. Their clean synergy in Clojure is *compelling*. We

Prepared exclusively for Michael Kevin Hagel

will cover all these features and more in Chapter 1, *Getting Started*, on page 13.

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 Java, C#, Ruby, or Python.

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 belongs in a general-purpose language. This book will appeal to three communities that Clojure is introducing to the mainstream: Lisp programmers, users of functional languages such as Haskell, and builders of concurrent systems.

Clojure is part of a larger phenomenon. Three other languages have also garnered attention recently for their support of functional programming and/or their concurrency model. Enthusiasts of Erlang, F#, and Scala should enjoy this book in comparison to their language of choice.

What Is In This Book

Chapter 1, *Getting Started*, on page 13 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 30 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.

Chapter 3, *Working with Java*, on page 57, shows you how to call Java from Clojure, and Clojure from Java. You will see how to take Clojure straight to the metal and get Java-level performance.

The next two chapters cover functional programming. Chapter 4, *Unifying Data with Sequences*, on page 77 shows how all data can be unified under the powerful sequence metaphor. Chapter 5, *Functional Program-*

ming, on page 105 shows you how to write functional code in the same style used by the sequence library.

Chapter 6, *Concurrency*, on page 106 delves into Clojure's concurrency model. Clojure provides three powerful models for dealing with concurrency, plus all of the goodness of Java's concurrency libraries.

Chapter 7, *Macros*, on page 123 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 124 covers Clojure's answer to polymorphism. Clojure's multimethods support arbitrary dispatch on any function of a method's arguments.

There is already a thriving Clojure community. Chapter 9, *Clojure-Contrib: Clojure's Standard Library*, on page 125 introduces some key third-party libraries for Clojure.

Chapter 10, *Case Study*, on page 126 ties things together, using concepts from throughout the book to build a real-world case study.

Appendix A, on page 127 cites other books referenced in the book.

Appendix B, on page 128 gives the signature for each Clojure form covered in the book, with reference to where each form is introduced in the main text.

Appendix C, on page 132 lists editor support options for Clojure, with links to setup instructions for each.

Appendix D, on page 134 describes the Monte Carlo simulation for the value of π that we parallelize in Chapter 6, *Concurrency*, on page 106.

How to Read This Book

Start by reading the first two chapters in order. Pay particular attention to Section 1.2, *Why Clojure?*, on page 14, which provides an overview of Clojure's advantages.

Experiment continuously. Clojure provides an interactive environment where you can get immediate feedback, see Section 1.3, *Using the REPL*, on page 22 for more information.

After you read the first two chapters, skip around as you like. But read Chapter 4, *Unifying Data with Sequences*, on page 77 before you read

Prepared exclusively for Michael Kevin Hagel

Chapter 6, *Concurrency*, on page 106. These chapters lead you from Clojure's immutable data structures to a powerful model for writing correct concurrency programs.

I do not cover Lisp indentation in the book. For the later chapters, make sure that you get an editor that does indentation for you, and you will quickly pick the rules. Appendix C, on page 132 will point you to common editor options.

For Lisp Programmers

- Some of Chapter 2, *Exploring Clojure*, on page 30 will be review, but read it anyway. Clojure introduces quite a bit of syntax (compared to many Lisps) and it is covered here.
- Pay close attention to the lazy data structures in Chapter 5, *Functional Programming*, on page 105.
- Get an Emacs mode for Clojure that makes you happy before working through the code examples in later chapters.

For Java Programmers

- Read Chapter 2, *Exploring Clojure*, on page 30 carefully. Clojure has very little syntax (compared to Java) and I cover the ground rules fairly quickly.
- Pay close attention to macros in Chapter 7, *Macros*, on page 123. These are the most alien part of Clojure, when viewed from a Java perspective.

Downloading Sample Code

You can download the sample code for the book from the Pragmatic Bookshelf web site.[2] The samples are organized as one directory per chapter. For example, the directory for Chapter 6, *Concurrency*, on page 106 is named concurrency. Within the sample directory for a specific chapter, there is typically a main sample file of the same name, for example introduction.clj.

If you are reading the book in PDF form, you can click on the little gray box preceding a code excerpt and download that excerpt directly.

2. http://www.pragprog.com/titles/shcloj

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.

Prepared exclusively for Michael Kevin Hagel

Chapter 1

Getting Started

In this chapter, you will see examples of Clojure's key features: elegance, Lisp syntax, direct Java interoperability, sequences, functional programming, and lock-free concurrency. We will end by putting these together in a complete sample program.

We will begin with just enough syntax to follow the examples; the rest you can pick up as you go.

1.1 Reading Clojure

If you do not already have a Lisp background, Clojure syntax takes some getting used to. But you can read quite a bit of Clojure by simply remembering these two rules:

- expressions begin with a verb
- entire expressions are parenthesized

Some comparisons to Java will make these rules clear. First, consider simple addition:

```
// java
2 + 2
```

```
; clojure
(+ 2 2)
```

As you can see, the verb + comes first, and there are parentheses around the entire expression. A parenthesized expression is called a *list*.

Second, consider invoking a series of Java methods:

Prepared exclusively for Michael Kevin Hagel

```
// java
"hello".getClass().getProtectionDomain().getPermissions()

; clojure
(.. "hello" getClass getProtectionDomain getPermissions)
```

Again, the verb .. comes first. The .. performs a *threaded member access*. It takes "hello", calls getClass(), then threads the result of that call to getProtectionDomain(), and so on.

Notice that the Clojure version has *markedly fewer parentheses* than the Java version. Clojure is a Lisp, and Lisp has been jokingly described as "Lost in a Sea of Parentheses."[1] And yet, here you can see that Clojure code has fewer parentheses than the equivalent Java.

Now that you can read a bit of Clojure, let's talk about why you will want to.

1.2 Why Clojure?

Clojure feels like a general-purpose language beamed back from the near future. Its support for functional programming and software transactional memory are well beyond current practice, and will surely become an important paradigm as hardware becomes more concurrent.

At the same time, Clojure is well-grounded in the past and the present. It brings together Lisp and the Java Virtual Machine. Lisp brings wisdom spanning most the history of programming, and Java brings the robustness and tooling of the dominant platform available today.

Let's explore this powerful combination.

Clojure is Elegant

Clojure is high signal, low noise. As a result, Clojure programs are short programs. Short programs are cheaper to build, cheaper to deploy, and cheaper to maintain. This is particularly true when the programs are concise, rather than merely terse. As an example, consider the following Java code, from the Apache Commons:

Download introduction/isBlank.java

```
public class StringUtils {
  public static boolean isBlank(String str) {
```

1. There are dozens of variants of the Lisp acronym joke, see http://www.abbreviations.com/LISP

Prepared exclusively for Michael Kevin Hagel

```
    int strLen;
    if (str == null || (strLen = str.length()) == 0) {
      return true;
    }
    for (int i = 0; i < strLen; i++) {
    if ((Character.isWhitespace(str.charAt(i)) == false)) {
      return false;
    }
  }
}
```

The isBlank() method checks to see if a string is *blank*: either empty or consisting of only whitespace. Here is a similar implementation in Clojure:

Download introduction/introduction.clj

```
(defmulti blank? class)
(defmethod blank? String [s] (every? #{\space} s))
(defmethod blank? nil [_] true)
```

The Clojure version is three times shorter (by lines of code) and much simpler (it has *no* branches). Two of Clojure's features contribute to making the code so simple:

1. Clojure's *multimethods* allow different implementations to be provided for different situations. This saves the trouble of introducing a meaningless class (StringUtils) and avoids the need for an **if** statement to handle the null case.

2. First class functions allow the expressive (every? #{\space} s), avoiding the need for a **for** loop over the string.

As another example, consider defining a trivial Person class in Java:

Download introduction/Person.java

```
public class Person {
        public Person(String firstName, String lastName) {
                this.firstName = firstName;
                this.lastName = lastName;
        }
        public String getFirstName() {
                return firstName;
        }
        public void setFirstName(String firstName) {
                this.firstName = firstName;
        }
        public String getLastName() {
                return lastName;
        }
        public void setLastName(String lastName) {
```

```
                this.lastName = lastName;
        }
}
```

In Clojure, you would define person with a single line:

```
(defstruct person :first-name :last-name)
```

Other than being an order of magnitude shorter, the Clojure approach differs in that a Clojure person is *immutable*. This helps to enable functional programming, covered in Chapter 5, *Functional Programming*, on page 105. Because structures are immutable, Clojure provides correct implementations of hashCode() and equals() automatically.

Clojure has a lot of elegance baked in, but if you find something missing you can add it yourself, thanks to the power of Lisp.

Clojure is Lisp Reloaded

Clojure is a Lisp. For decades, Lisp advocates have pointed out the advantages that Lisp has over, well, everything else. At the same time, Lisp's world domination plan seems to be proceeding slowly.

Like any other Lisp, Clojure faces two challenges:

- Clojure must succeed as a Lisp, by persuading Lisp programmers that Clojure embraces the critical parts of Lisp
- At the same time, Clojure must succeed *where past Lisps have failed*, by winning support from the broader community of programmers.

Clojure meets these challenges, by providing the meta capabilities of Lisp, and at the same time embracing a set of syntax enhancements that make Clojure friendly and accessible to non-Lisp programmers.

Why Lisp

Polymorphism provides a useful metaphor for understanding the power of Lisp. Consider the Person class in the previous section. It has a getFirstName() method that you can invoke like this:

```
somePerson.getFirstName()
```

The nice thing about getFirstName is that you don't have to know the specific implementation of somePerson. Maybe it is a Person, or maybe it is some subclass of Person. Java's polymorphism guarantees that the correct implementation will be selected based on the runtime type of somePerson.

Prepared exclusively for Michael Kevin Hagel

Polymorphism encourages reuse by breaking dependencies. Your call to getFirstName() can be reused with other implementations of Person, for free. You don't have to change or recompile your code.

Now, let's return for a moment to the definition of Person, narrowing our focus to just these two lines:

```
public class Person {
        public String getFirstName() {
// continues
```

Since Java is polymorphic, you should be able to plug in an alternate implementation of public, or of class, right? Wrong. Java methods are polymorphic, but Java *keywords* are not. They are defined by the language, and are fixed.

Why would you want to change the semantics of public or class anyway? It may sound crazy at first, but once you think about it there are plenty of reasons. If you could change keywords, you could do things like:

- Redefine private to mean "private for production code, but public for serialization and unit tests".
- Redefine class to automatically generate getters and setters for private fields, unless otherwise directed.
- Create a subclass of class that provides callback hooks for lifecycle events. For example, a lifecycle-aware class could fire an event whenever an instance of the class is created.

I have seen Java programs that needed all these features. Without them, programmers have to resort to repetitive, error-prone workarounds. Literally *millions* of lines of Java code have been written to work around the fixed behaviors of the core language.

Now, back to Clojure. Features that would be keywords in Java are often *macros* in Clojure. For example, defstruct is a macro:

```
(defstruct person :first-name :last-name)
```

Because it is a macro, defstruct() is itself written in Clojure. You can change or augment the semantics of defstruct() in your own programs. Or, you can write a different macro that has the semantics you need. If you want a variant of structs that requires non-null values for all fields, you can create something like:

```
(my-defstruct person {String :first-name, String :last-name}
  :allow-nulls false)
```

Prepared exclusively for Michael Kevin Hagel

This ability to reprogram the language from within the language is the unique advantage of Lisp. You will see this idea described in various ways:

- Code is data.
- The whole language there, all the time.

The downside of this ability, at least for beginners, is Lisp's fixation on parentheses, and on lists as the core data type. Clojure offers an interesting new approach that makes Lisp syntax more approachable for non-Lispers.

Lisp, with Fewer Parentheses

Newcomers to Lisp are intimidated by the parentheses. That's too bad, because the parentheses are central to Lisp's power. Clojure keeps the parentheses (and the power of Lisp!) but it improves on traditional Lisp syntax in several ways:

- Clojure generalizes Lisp's physical list into an abstraction called a *sequence*. Clojure provides a convenient literal syntax for a wide variety of sequences: regular expressions, maps, sets, vectors, and metadata, just to name a few. This makes Clojure code less list-y than most Lisps. For example, function arguments are specified in a vector ([]) instead of a list (()):

  ```
  (defn hello-world [name]
    (println (format "Hello, %s") name)
  ```

  The vector makes the argument list jump out visually, and makes Clojure function definitions easy to read.

- In Clojure, commas are whitespace. Lisp doesn't care about commas anyway, and adding commas can make some data structures more readable. Consider vectors:

  ```
  ; make vectors look like arrays in other languages
  user=> [1, 2, 3, 4]
  [1 2 3 4]
  ```

- Idiomatic Clojure does not nest parentheses more than necessary. Consider dotimes macro, present in both Common Lisp and Clojure. As its name suggests, dotimes does something a certain count of times, making the iteration available in a variable. The iteration variable and the count are related, so Common Lisp groups them with parentheses:

```
(dotimes (x 10) ...)
```

Clojure avoids these parentheses:

```
(dotimes x 10 ...)
```

This is an aesthetic decision, and both approaches have their supporters. The important thing is that Clojure takes the opportunity to be less Lisp-y whenever it can do so without compromising Lisp's power.

Clojure is a great Lisp for non-Lisp programmers.

Clojure is a Functional Language

Clojure is a functional language. In general, this means that:

- Functions are first-class objects.
- Data is immutable.
- Functions are *pure* (they have no side effects).

In practice, this leads to code that has no side effects, no variables, and no loops. Such code is easier to understand, and much easier to reuse. For example, the following short program searches a database of compositions for the composer for all compositions named "Requiem":

```
(for [c compositions :when (= "Requiem" (:name c))] (:composer c))
-> ("W. A. Mozart" "Guiseppe Verdi")
```

This code has four desirable properties:

- It is *simple*: it has no loops, variables or mutable state.
- It is *thread-safe*: no locking is needed.
- It is *parallelizable*: you could farm individual steps out to multiple threads without changing the code for each step.
- It is *generic*: compositions could be a plain set, or XML, or a database result set.

A typical object-oriented approach would have none of these properties. Instead, it would be complex, not thread safe, not parallelizable, and specific to one inheritance hierarchy.

People have known about the advantages of functional languages for a while now. And yet, pure functional languages like Haskell have not taken over the world, because developers find that not everything fits the pure functional view.

Prepared exclusively for Michael Kevin Hagel

There are three reasons that Clojure can attract more interest now than functional languages have in the past:

- Clojure's Java invocation approach is *not* functional. When you call Java, you enter the familiar, mutable world. This offers a comfortable reprieve for beginners learning functional programming.
- Functional programming is more urgent today than ever before. Massively parallel hardware is right around the corner, and functional languages provide a clear approach for taking advantage of it.
- Purely functional languages have difficulty modeling state that really needs to change. Clojure provides a structured mechanism for working with changeable state via software transactional memory, the agent system, and the dynamic variable system.

Clojure's approach to changing state enables concurrency without locking, and complements Clojure's functional core.

Clojure is Concurrent

Clojure's support for functional programming makes it easy to write thread-safe code. Since immutable data structures cannot *ever* change, there is no danger of data corruption based on another thread's activity.

However, Clojure's support for concurrency goes beyond just functional programming. When you need mutable, cross-thread references to data, Clojure provides them via software transactional memory (STM). STM is a higher-level approach to thread-safety than the locking mechanisms that Java provides. Rather than creating fragile, error-prone locking strategies, you can protect shared state with transactions. This is much more productive, as many programmers have a good understanding of transactions based on experience with databases.

For example, the code below creates a working, thread-safe, in-memory database of accounts:

```
(def accounts (ref #{}))
(defstruct account :id :balance)
```

The ref() function creates a transactionally protected reference to the current state of the database. Updating is trivial:

```
(dosync (alter accounts conj (struct account "CLJ" 1000.00)))
```

The dosync() causes the update to accounts to execute inside a transaction. Thread safety is guaranteed, and readers never block.

While the example here is trivial, the technique is general, and works on real-world-sized problems. See Chapter 6, *Concurrency*, on page 106 for more on concurrency and STM in Clojure.

Clojure Embraces the Java Virtual Machine

Clojure gives you clean, simple, direct access to Java. You can call any Java API directly:

```
(System/getProperties)
-> {java.runtime.name=Java(TM) SE Runtime Environment
    ... many more ...
```

Clojure adds a lot of syntactic sugar for calling Java. I won't get into the details here (see Section 3.1, *Calling Java*, on page 58), but notice that in the following code the Clojure version has both fewer dots *and fewer parentheses* than the Java version:

```
// Java
"hello".getClass().getProtectionDomain().getCodeSource()

; Clojure
(.. "hello" getClass getProtectionDomain getCodeSource)
```

Clojure also provides simple functions for implementing Java interfaces and subclassing Java classes. Wherever possible, these functions are called for you automatically. For example, the code below automatically wraps a Clojure function in a Runnable so that it can run on a separate thread.

```
(.start (Thread. (fn [] (println "Hello" (Thread/currentThread)))))
-> Hello Thread[Thread-1,5,main]
```

Because the Java invocation syntax in Clojure is clean and simple, it is idiomatic to use Java directly, rather than to hide Java behind Lispy wrappers.

Now that you have seen a few of the reasons to use Clojure, it is time to roll up your sleeves and write some code.

1.3 Running Clojure

To run Clojure, you need two things:

- A Java runtime. Download and install Java (version 5 or greater) from http://java.sun.com/javase/downloads/index.jsp.

- Clojure. Clojure is hosted at SourceForge. Download the most recent clojure.jar.[2]

Important Note: Clojure is changing rapidly as I write the book. If you are reading during the Beta Book period, you will need to build Clojure from source:

```
svn co https://clojure.svn.sourceforge.net/svnroot/clojure clojure
cd clojure/trunk
ant jar
```

Once you have downloaded clojure, you can test your install by navigating to the directory where you placed clojure.jar, and running the Clojure *Read-Eval-Print Loop* (REPL). The REPL should prompt you with user=>:

```
java -cp clojure.jar clojure.lang.Repl
Clojure
user=>
```

Now you are ready for "Hello world."

Using the REPL

To see how to use the REPL, let's create a few variants of "Hello world." First, type (println "hello world") at the REPL prompt:

```
user=> (println "hello world")
hello world
nil
```

The second line, hello world, is the console output you requested. This third line, nil, is the return value of the call to println().

Next, encapsulate your "Hello World" into a function that can address a person by name:

```
user=> (defn hello [name] (str "Hello, " name))
#=(var user/hello)
```

Let's break this down:

- defn defines a function.
- hello is the function name.
- hello takes one argument, name.
- str concatenates an arbitrary list of arguments into a string.

2. http://sourceforge.net/project/showfiles.php?group_id=137961

Prepared exclusively for Michael Kevin Hagel

- The return value, user/hello, is the fully-qualified name of the function. The user namespace is the REPL default, like the default package in Java.

Now you can call hello, passing in your name:

```
user=> (hello "Stu")
Hello, Stu
```

The REPL includes several useful special variables. The results of the last three evaluations are included in the special variables *1, *2, *3, respectively. This makes it easy to work iteratively. Say hello to a few different names:

```
user=> (hello "Stu")
Hello, Stu

user=> (hello "Clojure")
Hello, Clojure
```

Now, you can use the special variables to combine results of your recent work:

```
(str *1 " and " *2)
-> "Hello, Clojure and Hello, Stu"
```

If you make a mistake in the REPL, you will see a Java exception. The details are often omitted for brevity. For example, dividing by zero is a no-no:

```
user=> (/ 1 0)
java.lang.ArithmeticException: Divide by zero (NO_SOURCE_FILE:0)
```

Here the problem is obvious, but sometimes the problem is more subtle and you want the detailed stack trace. No problem. The *e special variable holds the last exception. Because Clojure exceptions are Java exceptions, you can call Java methods such as printStackTrace():

```
user=> (.printStackTrace *e)
java.lang.ArithmeticException: Divide by zero (NO_SOURCE_FILE:0)
  at clojure.lang.Compiler.eval(Compiler.java:4094)
  at clojure.lang.Repl.main(Repl.java:87)
Caused by: java.lang.ArithmeticException: Divide by zero
  at clojure.lang.Numbers.divide(Numbers.java:142)
  at user.eval__2677.invoke(Unknown Source)
  at clojure.lang.Compiler.eval(Compiler.java:4083)
  ... 1 more
```

Java interop is covered in Chapter 3, *Working with Java*, on page 57.

If you have a block of code that is too large to conveniently type at the REPL, save the code into a file, and then load that file from the REPL. You can use an absolute path, or a path relative to where you launched the REPL:

```
; save some work in temp.clj, and then ...
user=> (load-file "temp.clj")
```

The REPL is a terrific environment for trying ideas and getting immediate feedback. For best results, keep a REPL open at all times while reading this book.

Adding Shared State

The hello function of the previous section is *pure*, that is, it has no side effects. Pure functions are easy to develop, test, and understand, and you should prefer them for many tasks.

That said, most programs have some shared state, and will need impure functions to manage that shared state. Let's extend hello to keep track of past visitors, and offer a different greeting to people it has met before. First, you will need something to track the visitors. A set will do the trick:

```
#{}
-> #{}
```

The #{} is a literal for an empty set, since there are no visitors yet.

You can use

```
(conj coll item)
```

to build a new set with an element added. conj is short for conjoin. conj an element onto a set to see that a new set is created:

```
(conj #{} "stu")
#{"stu"}
```

Now that you can build new sets, you need some way to keep track of the *current* set of visitors. Clojure provides *references* (refs) for this purpose:

```
(def visitors (ref #{}))
#=(var user/visitors)
```

In the code above, ref creates a reference to a (currently empty) set, and def binds the name visitors to the ref.

Now you can create a transaction that updates the ref whenever a new visitor comes along:

```
(dosync (commute visitors conj "stu"))
#{"stu"}
```

dosync creates a transaction. Inside the transaction, commute updates the visitors ref. commute calls a commutative function on the contents of a ref, that is, a function where ordering doesn't matter. conj is commutative for sets. (When you add items to a set, it doesn't matter which item you add first.) commute is part of Clojure's strategy for minimizing contention in transactions, and is discussed in Section 6.1, *commute*, on page 108.

At any time, you can peek inside the ref with deref, or with the shorter @:

```
(deref visitors)
-> #{"Stu"}

user=> @visitors
-> #{"Stu"}
```

Now you are ready to build the new, more elaborate version of hello:

Download introduction/introduction.clj

```
(defn hello [name]
  (let [past-visitor (@visitors name)]
    (dosync (commute visitors conj name))
    (if past-visitor
      (str "Welcome back, " name)
      (str "Hello, " name))))
```

On 2, hello checks to see if name is already in the set that visitors refers to, remembering the result in past-visitor.

On 3, commute updates the visitors to include the name name.

Lines 4 through 6 return different strings based on the the user was a visitor in the past.

You can verify that new visitors get one message the first time around. . .

```
(hello "world")
-> "Hello, world"
```

. . . And a different message when they return again.

```
(hello "world")
-> "Welcome back, world"
```

The use of references and transactions above offers great benefits: The hello function is safe for multiple threads and processors. This is true, even though there are *no locks*, either directly in the code, or hidden

down inside Clojure. As a result, hello is also fast *and scalable* across multiple processors. Clojure transactions are described in more detail in Chapter 6, *Concurrency*, on page 106.

At this point you should feel comfortable entering small bits of code at the REPL. Larger units of code aren't that different; you can load and run them from the REPL as well. Let's explore that next.

1.4 Using the Sample Code

Before we go any further, it is important to make sure that you can browse *and execute* the code examples that accompany the book. Follow the instructions in Section , *Downloading Sample Code*, on page 11 to download the sample code.

One you have downloaded the samples, the best way to execute them is interactively from the REPL. Open up a REPL, making sure you have the following items on the CLASSPATH:

- the Clojure runtime: clojure.jar
- Clojure's "standard library"clojure-contrib.jar
- the root directory for the book samples

For example:

```
export CLASSPATH=/jars/clojure.jar:\
/jars/clojure-contrib.jar:\
/book/examples
java clojure.lang.Repl
```

The program should respond with the Clojure REPL:

```
Clojure
user=>
```

If you see the Clojure REPL, then java was able to find clojure.jar.

Next, test that you can use clojure-contrib.jar by requiring a library from Clojure Contrib. A Clojure library has a name like x.y.some-lib. When you (require 'x.y.some-lib), Clojure looks for a file named x/y/some_lib/some_lib.clj on the CLASSPATH. Try requiring clojure.contrib.str-utils:

```
user=> (require :verbose 'clojure.contrib.str-utils)
nil
```

If you have not correctly specified the path to clojure-contrib.jar, you will see a message like this instead:

```
user=> (require 'clojure.contrib.str-utils)
java.io.FileNotFoundException: Could not locate Clojure resource
on classpath: clojure/contrib/str_utils/str_utils.clj
```

Finally, test that you can load the sample code for the book:

```
user=> (require 'introduction)
nil
```

The introduction library includes an implementation of the Fibonacci numbers, which is the traditional "Hello, World" program for functional languages. We will explore the Fibonacci numbers in more detail in Chapter 5, *Functional Programming*, on page 105. For now, just make sure that you can execute the sample function introduction.fibs. Enter the following line of code at the REPL to take the first 10 Fibonacci numbers.

```
user=> (take 10 introduction.fibs)
(0 1 1 2 3 5 8 13 21 34)
```

If you see the first ten Fibonacci numbers as listed above, you have successfully installed the book samples.

Formatting Examples at the REPL

The REPL prompt includes the current namespace, initially user. For most of the book's examples, the current namespace is irrelevant. Where the namespace is irrelevant, I will use the following syntax for interaction with the REPL:

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

Where the current namespace is important, I will use:

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

Don't Just Require, Use!

When you require a Clojure library, you must refer to items in the library with a namespace-qualified name. Instead of fibs you must say introduction.fibs:

```
(require 'introduction)
-> nil

(take 10 introduction.fibs)
-> (0 1 1 2 3 5 8 13 21 34)
```

For convenience, the use function will require a library, and make the public names in the library available to the current namespace. Unless you have a name collision, you can use libraries directly, without namespace qualification:

```
(use 'introduction)
-> nil

(take 10 fibs)
-> (0 1 1 2 3 5 8 13 21 34)
```

Except where specifically noted, the book examples contain no namespace collisions. You should be able to use all of the chapter samples from the same REPL.

Reloading Changed Libraries

As you are working through the book samples, you can add the :reload flag to require or use to force a library to reload. This is useful if you are making changes and want to see results without restarting the REPL:

```
(use :reload 'introduction)
-> nil
```

The :reload flag worked, but the nil return value of use does not tell you what happened. For more information, you can add the :verbose flag. If you require a library what is already loaded, verbose is silent:

```
(require :verbose 'introduction)
-> nil
```

If you force Clojure to :reload the introduction library, :verbose will show you that the file is loaded, and that the introduction namespace is referred so that you can use unqualified names.

```
(use :reload :verbose 'introduction)
(clojure/load "/introduction/introduction.clj")
(clojure/in-ns 'user)
(clojure/refer 'introduction)
-> nil
```

Of you are having trouble with require and use, there are other ways to reload examples. You can always restart the REPL, or explicitly load code by passing a pathname to load-file.

1.5 Wrapping Up

You have just gotten the whirlwind tour of Clojure. You have seen Clojure's expressive syntax, learned about Clojure's approach to Lisp, and

seen how easy it is to call Java code.

You have Clojure running in your own environment, and you have written short programs at the REPL to demonstrate functional programming and software transactional memory. Now it is time to explore the entire language.

Prepared exclusively for Michael Kevin Hagel

Chapter 2

Exploring Clojure

Clojure offers great power through clean Java interop, functional style, and concurrency support. But before you can appreciate all these, you have to start with the language basics. In this chapter you will take a quick tour of the Clojure language, including:

- forms
- reader macros
- functions
- bindings and namespaces
- flow control
- metadata

If your background is primarily in imperative languages such as Java, this tour may seem to be missing key language constructs, including basic things like variables and **for** loops. The section Section 2.7, *Where's My For Loop?*, on page 52 will show you how you can *live better* without **for** loops and variables.

Clojure is very expressive, and this chapter covers many concepts quickly. Don't worry if you don't understand every detail; we will revisit these topics in more detail in later chapters. If possible, bring up a REPL, and follow along with the examples as you read.

2.1 Forms

Clojure is *homoiconic*, which is to say that Clojure code is composed of Clojure data. When you run a Clojure program, a part of Clojure

Prepared exclusively for Michael Kevin Hagel

called the *reader* reads the text of the program in chunks called *forms* and translates them into Clojure data structures. Clojure then takes executes the forms.

To see forms in action, let's start with some simple forms supporting numeric types.

Using Numeric Types

Numeric literals are forms. Numbers simply evaluate to themselves. If you enter a number, the REPL will give it back to you:

```
42
-> 42
```

A list of numbers is another kind of form. Create a list of the numbers 1,2, and 3:

```
'(1 2 3)
-> (1 2 3)
```

Notice the quote in front of the list. This quote tells Clojure "do not evaluate what comes next, just return it." The quote is necessary because lists are special in Clojure. When Clojure evaluates a list, it tries to interpret the first element of this list as a function (or macro) and the remainder of the list as arguments. So, what happens if you leave the quote out?

```
user=> (1 2 3)
java.lang.ClassCastException: java.lang.Integer cannot be cast \
to clojure.lang.IFn
```

Clojure is trying (and failing) to interpret `1` as the name of a function. Next, try a list whose first item *is* a Clojure function, like the symbol +:

```
user=> (+ 1 2)
3
```

The style of placing the function first is called *prefix notation*, as opposed to the more familiar *infix notation* `1 + 2 = 3`. One advantage of prefix notation is that you can easily extend it for arbitrary numbers of arguments:

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

Even the degenerate case of no arguments works as you would expect, returning zero.

```
user=> (+)
0
```

Zero is the identity for addition, that is, including a zero in an addition does not affect the outcome. Thus a no-argument addition returns zero.

Symbols such as + are a kind of form, and are used to name things. For example, + names the function that adds things together. Symbols cannot start with a number, and can consist of alphanumeric characters, plus +, -, *, /, !, ?, ., and _.[1]

Many mathematical and comparison operators have the names and semantics that you would expect from other programming languages. Addition, subtraction, multiplication, comparison, and equality all work as you would expect:

```
(- 10 5)
-> 5

(* 3 10 10)
-> 300

(> 5 2)
-> true

(>= 5 5)
-> true

(< 5 2)
-> false

(= 5 2)
-> false
```

Division may surprise you:

```
(/ 22 7)
-> 22/7
```

As you can see, Clojure has a built-in Ratio type. If you actually want decimal division, use a floating-point literal for the dividend:

```
(/ 22.0 7)
-> 3.142857142857143
```

If you want to stick to integers, you can get the integer quotient and remainder with quot() and rem():

```
(quot 22 7)
-> 3
```

1. Clojure treats / and . specially in order to support namespaces, see Section 2.4, *Namespaces*, on page 44 for details.

Prepared exclusively for Michael Kevin Hagel

```
(rem 22 7)
-> 1
```

If you are doing arbitrary-precision math, append M to a number to create a BigDecimal literal:

```
(+ 1 (/ 0.00001 1000000000000000000))
-> 1.0

(+ 1 (/ 0.00001M 1000000000000000000))
-> 1.000000000000000000000001M
```

As the code above shows, doubles can lose precision, but BigDecimals cannot.

Clojure's approach to arbitrary-sized integers is simple: Just don't worry about it. Clojure will upgrade to BigInteger when you need it. Try creating some small and large integers, and then inspect their class.

```
(def small-int (* 1000 1000 1000))
-> #=(var myapp/small-int)

(class small-int)
-> #=java.lang.Integer

(def big-int (* 1000 1000 1000 1000 1000 1000))
-> #=(var myapp/big-int)

(class big-int)
-> #=java.math.BigInteger
```

Clojure math is also *fast*. See Section 3.2, *Optimizing for Performance*, on page 64 for details.

Strings and Characters

Strings are another kind of reader form. Clojure strings are Java strings. They are delimited by ", and they can span multiple lines:

```
"This is a\nmultiline string"
-> "This is a\nmultiline string"

"This is also
a multiline String"
-> "This is also\na multiline String"
```

Clojure does not wrap most of Java's string functions. Instead, you can call them directly using Clojure's Java interop forms:

```
(.toUpperCase "hello")
-> "HELLO"
```

The dot before toUpperCase tells Clojure to treat it as the name of a Java method instead of a Clojure function.

There is one exception: You do not need to call toString() directly. Instead of calling toString(), use Clojure's str function:

```
(str 1 2 nil 3)
-> "123"
```

The example above demonstrates str's advantages over toString(). It smashes together multiple arguments, and it skips nil without error.

Clojure characters are Java characters. Their literal syntax is \{letter}, where letter can be a letter, or newline, space, or tab.

Strings are sequences of characters. When you call Clojure sequence functions on a String, you get a sequence of characters back. Imagine that you wanted to conceal a secret message by interleaving it with a second, innocuous message. You could use interleave to combine the two messages:

```
(interleave "Attack at midnight" "The purple elephant chortled")
-> (\A \T \t \h \t \e \a \space \c \p \k \u \space \r
    \a \p \t \l \space \e \m \space \i \e \d \l \n \e
    \i \p \g \h \h \a \t \n)
```

That works, but you probably want the resulting sequence as a string for transmission. You can use (apply str ...) to build a string from a sequence of characters:

```
(apply str (interleave "Attack at midnight"
                       "The purple elephant chortled"))
-> "ATthtea cpku raptl em iedlneipghhatn"
```

You can use (apply str ...) again to reveal the message.

```
(apply str (take-nth 2 "ATthtea cpku raptl em iedlneipghhatn"))
-> "Attack at midnight"
```

The call to (take-nth 2 ...) takes every other element of the sequence, extracting the obfuscated message.

Booleans and Nil

Clojure's rules for booleans are easy to understand:

- true is true and false is false.
- Other than false, nil also evaluates to false when used in a boolean context.

- Other than false and, nil, *everything else* evaluates to true in a boolean context.

Lisp programmers be warned: the empty list is not false in Clojure:

```
;           (if part)              (else part)
(if '() "We are in Clojure!" "We are in Common Lisp!")
-> "We are in Clojure!"
```

Clojure includes a set of predicates for testing true?, false?, and nil?. Be careful using these. true? tests whether a value is actually true, not whether the value evaluates to true in a boolean context. The only thing that is true? is true:

```
(true? true)
-> true

(true? "foo")
-> false
```

Be careful with predicates like true? when filtering collections. Imagine that you need to filter all of the nil or false values out of a collection. You might try to pass true? as a predicate to filter:

```
(filter true? [1 1 2 false 3 nil 5])
-> nil
```

Oops. None of those values are literally true. What you need is a function that will return the value directly, and then let filter's boolean context interpret the value as true. identity of a value simply returns that value, and does the trick:

```
(identity 1)
-> 1

(filter identity [1 1 2 false 3 nil 5])
-> (1 1 2 3 5)
```

nil? and false? work the same way. Only nil? is nil, and only false is false?.

So far, you have seen numeric literals, lists, symbols, strings, characters, booleans, and nil. The remaining forms are covered later in the book, as they are needed. For your reference, Figure 2.1, on the next page lists all the forms, with a quick example to get you started, and a pointer to more complete coverage.

Prepared exclusively for Michael Kevin Hagel

Form	Example(s)	Primary Coverage
Boolean	true, false	Section 2.1, *Booleans and Nil*, on page 34
Character	\a	Section 2.1, *Strings and Characters*, on page 33
Keyword	:tag, :doc	Section 2.1, *Forms*, on page 30
List	(1 2 3), (println "foo")	Chapter 4, *Unifying Data with Sequences*, on page 77
Map	{:name "Bill", :age 42}	Chapter 4, *Unifying Data with Sequences*, on page 77
Nil	nil	Section 2.1, *Booleans and Nil*, on page 34
Number	1, 4.2	Section 2.1, *Using Numeric Types*, on page 31
Set	#{:snap :crackle :pop}	Chapter 4, *Unifying Data with Sequences*, on page 77
String	"hello"	Section 2.1, *Strings and Characters*, on page 33
Symbol	user/foo, java.lang.String	Section 2.1, *Using Numeric Types*, on page 31
Vector	[1 2 3]	Chapter 4, *Unifying Data with Sequences*, on page 77

Figure 2.1: Clojure Forms

2.2 Reader Macros

Clojure forms are read by the *reader*, which converts text into Clojure data structures. In addition to the basic forms, the Clojure reader also recognizes a set of *reader macros*.[2] Reader macros are special reader behaviors triggered by prefix *macro characters*.

The most familiar reader macro is the comment. The macro character that triggers a comment is ;, and the special reader behavior is "ignore everything else up to the end of this line."

Many reader macros are abbreviations of longer list forms, and are used to reduce clutter. You have already seen one of these. The quote character ' prevents evaluation:

```
'(1 2)
-> (1 2)
```

'(1 2) is equivalent to the longer (quote (1 2)):

```
(quote (1 2))
-> (1 2)
```

The other reader macros are covered later in the book. Figure 2.2, on the following page provides a quick syntax overview and references to where each reader macro is covered.

2. Reader macros are totally different from macros, which are discussed in Chapter 7, *Macros*, on page 123.

Prepared exclusively for Michael Kevin Hagel

Reader Macro	Example(s)	Primary Coverage
Comment	; single line comment	Section 2.2, *Reader Macros*, on the prev
Deref	@form => (deref form)	Chapter 6, *Concurrency*, on page 106
Meta	^form => (meta form)	Section 2.6, *Metadata*, on page 50
Metadata	#^metadata form => (with-meta form)	Section 2.6, *Metadata*, on page 50
Quote	'form => (quote form)	Section 2.1, *Forms*, on page 30
Regex Pattern	#"foo" => a java.util.regex.Pattern	Section 4.4, *Seq-ing Regular Expressions*
Syntax-quote	`x	Chapter 7, *Macros*, on page 123
Unquote	~	Chapter 7, *Macros*, on page 123
Unquote-splicing	~@	Chapter 7, *Macros*, on page 123
Var-quote	#'x => (var x)	Chapter 6, *Concurrency*, on page 106

Figure 2.2: Reader Macros

2.3 Functions

In Clojure, a function call is simply a list whose first element resolves to a function. For example, this call to str concatenates its arguments to create a string:

```
(str "hello" " " "world")
-> "hello world"
```

Function names are typically hyphenated, as in clear-agent-errors. If a function is a predicate, then by convention its name should end with a question mark. As an example, the predicates below test the type of their argument, and all end with a ?:

```
user=> (string? "hello")
true

user=> (keyword? :hello)
true

user=> (symbol? :hello)
true
```

To define your own functions, use defn:

```
(defn name doc-string? attr-map? [params*] body)
```

The attr-map adds metadata to the function, and is covered separately in Section 2.6, *Metadata*, on page 50. To demonstrate the other components of a function definition, create a greeting function that takes a name, and returns a greeting preceded by "Hello":

Download exploring/exploring.clj

```
(defn greeting
  "Returns a greeting of the form 'Hello, name.'"
  [name]
  (str "Hello, " name))
```

You can call greeting:

```
(greeting "world")
-> "Hello, world"
```

You can also consult the documentation for greeting:

```
(doc greeting)
-------------------------
exploring/greeting
([name])
  Returns a greeting of the form 'Hello, name.'
```

What does greeting do if the caller omits name?

```
(greeting)
-> java.lang.IllegalArgumentException: \
   Wrong number of args passed to: greeting (NO_SOURCE_FILE:0)
```

Clojure functions enforce their *arity*, that is, their expected number of arguments. If you call a function with an incorrect number of arguments, Clojure will throw an IllegalArgumentException. If you want to make greeting issue a generic greeting when the caller omits name, you can use this alternate form of defn, which takes multiple argument lists and method bodies:

```
(defn name doc-string? attr-map?
 ([params*] body)+ )
```

Different arities of the same function can call one another, so you can easily create a zero-argument greeting that delegates to the one-argument greeting, passing in a default name:

Download exploring/exploring.clj

```
(defn greeting
  "Returns a greeting of the form 'Hello, name.'
   Default name is 'world'."
  ([] (greeting "world"))
  ([name] (str "Hello, " name)))
```

You can verify that the new greeting works as expected:

```
(greeting)
-> "Hello, world"
```

You can create a function with variable arity by including an ampersand in the parameter list. Clojure will bind the name after the ampersand to a list of all the remaining parameters. The following function allows two people to go on a date with a variable number of chaperones:

Download exploring/exploring.clj

```
(defn date [person-1 person-2 & chaperones]
  (println person-1 "and" person-2
           "went out with" (count chaperones) "chaperones."))
```

```
(date "Romeo" "Juliet" "Friar Lawrence" "Nurse")
Romeo and Juliet went out with 2 chaperones.
```

Variable arity is very useful in recursive definitions, see Chapter 5, *Functional Programming*, on page 105 for examples.

Anonymous Functions

In addition to named functions with defn, you can also create anonymous functions with fn. There are at least three reasons to create an anonymous function:

- The function is so brief and self-explanatory that giving it a name makes the code harder to read, not easier.
- The function is only being used from inside another function, and needs a local name, not a top-level binding.
- The function is created inside another function, for the purpose of closing over some data.

Filter functions are often brief and self-explantory. For example, imagine that you want to create an index for a sequence of words, and you do not care about words shorter than three characters. You can write an indexable-word? function like this:

Download exploring/exploring.clj

```
(defn indexable-word? [word]
  (> (count word) 2))
```

Then, you can use indexable-word? to extract the indexable words from a sentence:

```
(use 'clojure.contrib.str-utils)  ; for re-split
(filter indexable-word? (re-split #"\\W+" "A fine day it is"))
-> ("fine" "day")
```

When to Use Anonymous Functions

Anonymous functions have a terse syntax that is not always appropriate. For the first example in Section 2.3, *Anonymous Functions*, on the previous page, you may actually prefer to be explicit and create the named function indexable-word?. That is perfectly fine, and will certainly be the right choice if indexable-word? needs to be called from more than once place.

Anonymous forms are an option, not a requirement. Use the anonymous forms only when you find that they make your code more readable. They take a little getting used to, so don't be surprised if you gradually use them more and more.

The call to re-split breaks the sentence into words, and then filter calls indexable-word? once for each word, returning those words where indexable-word? returns true.

Anonymous functions let you do the same thing in a single line. The syntax for an anonymous function is:

```
(fn [params*] body)
```

With this syntax, you can plug the implementation of indexable-word? directly into the call to filter:

```
(filter (fn [w] (> (count w) 2)) (re-split #"\\W+" "A fine day"))
-> ("fine" "day")
```

There is an ever shorter syntax for anonymous functions, using implicit parameter names. The parameters are named %1, %2, etc., or just % if there is only one. This syntax looks like

```
#body
```

You can rewrite the call to filter with the shorter anonymous form:

```
(filter #(> (count %) 2) (re-split #"\\W+" "A fine day it is"))
-> ("fine" "day")
```

A second motivation for anonymous functions is wanting a named function, but only inside the scope of another function. Continuing with the indexable-word? example, you could write:

Download exploring/exploring.clj

```
(defn indexable-words [text]
  (let [indexable-word? (fn [w] (> (count w) 2))]
```

```
    (filter indexable-word? (re-split #"\\W+" text))))
```

The let binds the name indexable-word? to the same anonymous function you wrote earlier, this time inside the (lexical) scope of indexable-words. (let is covered in more detail under Section 2.4, *Bindings and Namespaces*, on the following page.)

You can verify that indexable-words works as expected:

```
(indexable-words "a fine day it is")
-> ("fine" "day")
```

The combination of let and an anonymous function says to readers of your code: "The function indexable-word? is interesting enough to have a name, but is relevant only inside indexable-words."

A third reason to use anonymous functions is when you dynamically creating a function at runtime. Earlier, you implemented a simple greeting function. Extending this idea, you can create a make-greeter function that creates greeting functions. make-greeter will take a greeting-prefix, and return a new function that composes greetings from the greeting-prefix and a name.

Download exploring/exploring.clj

```
(defn make-greeter [greeting-prefix]
  (fn [name] (str greeting-prefix ", " name)))
```

It makes no sense to name the fn, as it is creating a *different* function each time make-greeter is called. However, you may want to name the results of specific calls to make-greeter. You can call def to name functions created by make-greeter:

```
(def hello-greeting (make-greeter "Hello"))
-> #=(var user/hello-greeting)

(def aloha-greeting (make-greeter "Aloha"))
-> #=(var user/aloha-greeting)
```

Now, you can call these functions, just like any other functions:

```
(hello-greeting "world")
-> "Hello, world"

(aloha-greeting "world")
-> "Aloha, world"
```

As you can see, the different greeter functions remember the value of greeting-prefix at the time they were created. More formally, the greeter functions are *closures* over the value of greeting-prefix.

Prepared exclusively for Michael Kevin Hagel

def and defn work by creating Vars and root bindings, as discussed in Section 2.4, *Bindings and Namespaces* and Section 6.3, *Managing Per-Thread State with Vars*, on page 117.

2.4 Bindings and Namespaces

A binding associates a name with a value. For example, in a function call, argument values bind to parameter names. In the call below, 10 binds to the name number inside the triple function:

```
(defn triple [number] (* 3 number))
-> #=(var user/triple)

(triple 10)
-> 30
```

A function's parameter bindings are *lexical*: They are visible only inside the text of the function body. Functions are not the only way to have create a lexical binding. The special form let does nothing other than create a set of lexical bindings:

```
(let [bindings*] exprs*)
```

Imagine that you want coordinates for the four corners of a square, given the bottom, left, and width. You can let the top and right coordinates, based on the values given.

Download exploring/exploring.clj

```
(defn square-corners [bottom left width]
  (let [top (+ bottom width)
        right (+ left width)]
    [[bottom left] [top left] [top right] [bottom right]]))
```

The let binds top and right. This saves you the trouble of calculating top and right more than once. (Both are needed twice to generate the return value.)

Root Bindings

You can call def to create a *root binding* that is visible everywhere in an application. One use for root bindings is capturing constants:

```
(def max-grade 100)
-> #=(var user/max-grade)

max-grade
-> 100
```

Root bindings also bind names to functions. When you call defn, it uses def internally. So function names like triple below are root bindings.

```
(defn triple [x] (* 3 x))
-> (var user/triple)
```

Root bindings can be rebound on a per-thread basis. This supports Lisp-style dynamic variables, and Aspect-Oriented Programming (AOP) techniques such as the wormhole pattern.[3] Dynamic rebinding is covered in Section 6.3, *Managing Per-Thread State with Vars*, on page 117.

Destructuring

In many programming languages, you bind a variable to an *entire* collection when you only need to access *part* of the collection.

Imagine that you are working with a database of book authors. You track both first and last name, but some functions need to use only the first name:

Download exploring/exploring.clj

```
(defn greet-author-1 [author]
  (println "Hello," (:first-name author)))
```

The greet-author-1 functions works fine:

```
(greet-author-1 {:last-name "Vinge" :first-name "Vernor"})
-> Hello, Vernor
```

The binding of author is unsatisfying. You don't need the author, all you need is the first name. Clojure solves this with *destructuring*. Any place that you bind names, you can nest a vector or a map in the binding to reach into the collection and bind only the part you want. Here is a variant of greet-author that binds only the first name:

Download exploring/exploring.clj

```
(defn greet-author-2 [{name :first-name}]
  (println "Hello," name))
```

The binding form {name :first-name} tells Clojure to bind name to the first-name of the function parameter. greet-author-2 behaves just like greet-author-1:

```
(greet-author-1 {:last-name "Vinge" :first-name "Vernor"})
-> Hello, Vernor
```

3. See *AspectJ in Action* [Lad03] for a description of wormhole and other AOP patterns

Just as you can use a map to destructure any key/value collection, you can use a vector to destructure any list:

```
(let [x [1 2 3]] x)
-> [1 2 3]

(let [[_ _ x] [1 2 3]] x)
3
```

The first expression above does no destructuring, so x binds to the entire vector [1 2 3]. The second expression destructures the collection, binding x to the third element. Binding the _ to the first and second elements is idiomatic for "I don't care about these bindings."

As a more realistic example, you can use destructuring to create an ellipsize function. ellipsize should take a string, and return the first three words followed by ....

Download exploring/exploring.clj

```
(use 'clojure.contrib.str-utils)  ; for re-split
(defn ellipsize [words]
  (let [[w1 w2 w3] (re-split #"\\s" words)]
    (str-join " " [w1 w2 w3 "..."])))

(ellipsize "The quick brown fox jumped over the lazy dog.")
-> "The quick brown ..."
```

re-split splits the string around whitespace, then the destructuring form [w1 w2 w3] grabs the first three words. The destructuring ignores any extra items, which is exactly what we want. Finally str-join reassembles the three words, adding the ellipsis at the end.

Namespaces

Root bindings live in a namespace. You can see this when you start the Clojure REPL and create a binding:

```
user=> (def foo 10)
#=(var user/foo)
```

The user=> prompt tells you that you are currently working in the user namespace.[4] You should treat user as a scratch namespace for exploratory development.

4. Most of the REPL session listings in the book omit the REPL prompt for brevity. In this section the REPL prompt will be included whenever the current namespace is important.

When Clojure resolves the name foo, it namespace-qualifies foo in the current namespace user. You can verify this by calling resolve to explicitly resolve the symbol foo:

```
(resolve 'foo)
-> #=(var user/foo)
```

You can switch namespaces with the ns macro. You don't even need to create the namespace first, Clojure will automatically do that for you. Try creating a myapp namespace:

```
user=> (ns myapp)
nil
myapp=>
```

Now you are in the myapp namespace, and anything you def or defn will belong to myapp.

When you create a new namespace, the java.lang package and the Clojure namespace are automatically available to you:

```
myapp=> String
#=java.lang.String

myapp=> #'doc
#=(var clojure/doc)
```

Classes outside java.lang must be fully-qualified. You cannot just say File:

```
(File/separator)
java.lang.Exception: No such namespace: File
```

Instead, you must specify the fully-qualified java.io.File

```
myapp=> (java.io.File/separator)
"/"
```

If you do not want to use a fully-qualified classname, you can *map* the classname into the current namespace using import:

```
(import '(package Class+))
```

Once you import a class, you can use its short name:

```
(import '(java.io File))
-> nil

myapp=> (File/separator)
-> "/"
```

Vars outside the current namespace must be either fully qualified or mapped into the current namespace. For example, str-join lives in clojure.contrib.str-utils:

```
(clojure.contrib.str-utils/str-join "." [127 0 0 1])
-> "127.0.0.1"

(str-join "." [127 0 0 1])
-> java.lang.Exception:
   Unable to resolve symbol: str-join in this context
```

To map str-join into the current namespace, call use on str-join's namespace:

```
(use 'clojure.contrib.str-utils)
-> nil

(str-join "." [127 0 0 1])
-> "127.0.0.1"
```

If you make changes in a namespace and want to make those changes available to a running program, add the :reload option to use.

```
(use :reload 'clojure.contrib.str-utils)
-> nil
```

I regularly :reload while working on code samples for this book. Each chapter has a primary samples file, which I continually modify and reload. For this chapter, you can load reload the examples at any time:

```
(use :reload 'exploring)
```

In Clojure, it is idiomatic to import Java classes and use namespaces at the top of a source file. The ns macro can contain calls to import and use, so that you can set up the namespace mappings in a single form at the top of a source file. For example, this call to ns appears at the top of the sample code for this chapter:

Download exploring/exploring.clj

```
(ns exploring
    (use utils clojure.contrib.seq-utils clojure.contrib.str-utils)
    (import (java.io File)))
```

Clojure's namespace functions can do quite a bit more than I have shown here. You can reflectively traverse namespaces, and add or remove mappings at any time. To find out more, issue this command at the REPL:

```
(find-doc "ns-")
```

Alternately, you can browse the documentation at http://clojure.org/namespaces.

2.5 Flow Control

Clojure has very few flow control forms. In this section you will meet if, do, and loop/recur. As it turns out, this is almost all you will ever need.

Branch with if

Clojure's if evaluates its first argument. If the argument is logically true, it returns the result of evaluating its second argument:

Download exploring/exploring.clj

```
(defn is-small? [number]
  (if (< number 100) "yes"))

(is-small? 50)
-> "yes"
```

If the second argument to if is logically false, it returns nil:

```
(is-small? 50000)
-> nil
```

If you want to define a result for the "else" part of if, add it as a third argument.

Download exploring/exploring.clj

```
(defn is-small? [number]
  (if (< number 100) "yes" "no"))

(is-small? 50000)
-> "no"
```

Notice that if looks no different from a Clojure function call. Both if and function calls consist of a list. But if is *not* a function. It cannot be. The rule for functions is "evaluate all the args, then apply the function to them." if does not follow this rule. If it did, it would always evaluate both the "in" and "else" forms, regardless of input.

In Lisp terminology if is called a *special form* because it has its own special-case rules for when its arguments get evaluated. In addition to the special forms built into Clojure, you can write your own special-case evaluation rules using macros. Macros are covered in Chapter 7, *Macros*, on page 123.

Macros are extremely powerful, because they make the entire language programmable. Most programming languages do not have this kind of flexibility, and flow-control constructs are baked in at the language level. This leads to bloat: Language designers have to be paranoid they didn't leave something out, because if they do, language users have no

way to fix the problem. Clojure can afford to have a small set of flow control forms, because you can use use macros to *add your own*.

Introduce Side Effects with do

Clojure's if allows only one form for each branch. What if you want to do more than one thing on a branch? For example, you might want to log that a certain branch was chosen. do takes any number of forms, evaluates them all, and returns the last. You can use a do to print a logging statement from within an if:

Download exploring/exploring.clj

```
(defn is-small? [number]
  (if (< number 100)
    "yes"
    (do
      (println "Saw a big number" number)
      "no")))

(is-small? 200)
Saw a big number  200
-> "no"
```

Printing a logging statement is an example of a *side effect*. The println does not contribute to the return value of is-small? at all. Instead, it reaches out into the world outside the function and actually *does something*.

Many programming languages mix pure functions and side effects in completely ad hoc fashion. Not Clojure. In Clojure, side effects are explicit and unusual. do is one way to say "side effects to follow." Since do ignores the return values of all its forms save the last, those forms must have side effects to be of any use at all.

Plan to use do rarely, and for side effects, not for flow control. For those occasions where you need more complex control flow than a simple if, you should define a recurrence with loop/recur.

Recur with loop/recur

The Swiss Army Knife of flow control in Clojure is loop:

```
(loop [bindings *] exprs*)
```

The loop special form works like let, establishing bindings and then evaluating exprs. The difference is that loop sets a recursion point, which can then be targeted by the recur special form:

```
(recur exprs*)
```

recur binds new values for loop's exprs, and returns control to the top of the loop. For example, the loop/recur below returns a countdown:

Download exploring/exploring.clj

```
(loop [result [] x 5]
  (if (zero? x)
    result
    (recur (conj result x) (dec x))))
```

```
-> [5 4 3 2 1]
```

The first time through, loop binds result an empty vector, and binds x to 5. The recur then binds result to the vector with x added, binds x to one less than the previous x, and transfers control to the top of loop. The if guarantees that the recursion will terminate.

Instead of using a loop, you can use recur back to the argument bindings at the top of a function:

Download exploring/exploring.clj

```
(defn countdown [result x]
  (if (zero? x)
    result
    (recur (conj result x) (dec x))))
```

```
(countdown [] 5)
-> [5 4 3 2 1]
```

recur is a powerful building block, and is in fact the only non-stack-consuming loop in Clojure. But you many not use it very often, as many common recursions are provided by Clojure's sequence library. For example, countdown could also be expressed as any of the following:

```
user=> (into [] (take 5 (iterate dec 5)))
-> [5 4 3 2 1]

user=> (into [] (drop-last (reverse (range 6))))
-> [5 4 3 2 1]

user=> (apply vector (reverse (rest (range 6))))
-> [5 4 3 2 1]
```

Don't expect these forms to make sense yet–just be aware that there are often alternatives to using recur directly. The sequence library functions used here are described in Section 4.2, *Using the Sequence Library*, on page 82.

try and throw work as in Java, and are covered in Section 3.4, *Exception Handling*, on page 74.

2.6 Metadata

The wikipedia begins by saying that metadata is "data about data."[5] That is true, but not usably specific. In Clojure, metadata is data that is *orthogonal to the logical value of an object*. For example, a person's first and last names are plain old data. The fact that a person object can be serialized to XML has nothing to do with the person, and is metadata. Likewise, the fact that a person object is dirty and needs to be pushed to the database is metadata.

You can add metadata to an object with the with-meta function:

```
(def stu {:name "Stu" :email "stu@thinkrelevance.com"})
(def serializable-stu (with-meta stu {:serializable true}))
```

Metadata makes no difference for operations that depend on an object's value, so stu and serializable-stu are equal:

```
(= stu serializable-stu)
-> true
```

You can access metadata with the meta macro, verifying that serializable-stu has metadata, and stu does not.

```
(meta stu)
-> nil

(meta serializable-stu)
-> {:serializable true}
```

For convenience, you do not even have to spell out the meta function. You can use the reader macro ^ instead:

```
^stu
-> nil

^serializable-stu
-> {:serializable true}
```

The with-meta form can become verbose when specifying metadata, so there is a shortcut. The metadata dispatch macro #^ causes the form immediately following it to become metadata for the next form. So you could also define serializable-stu like this:

```
(def serializable-stu #^{:serializable true} {:name "stu"})
```

When you create a new object based on an existing object, the existing object's metadata flows to the new object. Usually this is the behavior

5. http://en.wikipedia.org/wiki/Metadata

Prepared exclusively for Michael Kevin Hagel

you would want. For example, you could add some more information to serializable-stu:

```
user=> (def stu-with-address (assoc serializable-stu :state "NC"))
-> {:name "Stu", :email "stu@thinkrelevance.com", :state "NC"}
```

The metadata from serializable-stu flows to stu-with-address:

```
^stu-with-address
-> {:serializable true}
```

One important use for metadata is type information. You can use a :tag key to tell the compiler that a particular type is expected. You can create a shout function that first checks to see that message is really a String:

```
(defn shout [#^{:tag String} message] (.toUpperCase message))

-> (shout 10)
java.lang.ClassCastException:
java.lang.Integer cannot be cast to java.lang.String
```

The ClassCastException shows that Clojure casts message to a String*before* calling toUpperCase.

Type tagging is common enough that Clojure lets you use String as shorthand for {:tag String}. For example, here is the definition of subs from Clojure's boot.clj, declaring that its arguments are strings:

```
(defn subs
  ([#^String s start] (. s (substring start)))
  ([#^String s start end] (. s (substring start end))))
```

Internally, Clojure makes heavy use of metadata. For example, functions and macros attach metadata to track their origin (file and line), their documentation string, their argument lists, and their namespace. Thus you can ask the meta macro for its own metadata:

```
(meta #'meta)
-> {:arglists ([obj]), :name meta, :file "boot.clj",
    :line 147, :ns #=(find-ns clojure),
    :doc "Returns the metadata of obj, returns nil if there is no\
          metadata."}
```

You can create whatever metadata keys you like, and interpret them as you see fit. To avoid conflicts with others, put your metadata keywords into a namespace:

```
(def persistent-stu #^{:com.thinkrelevance/persistent true} stu)
-> #=(var user/persistent-stu)
```

Prepared exclusively for Michael Kevin Hagel

The example above demonstrates namespace conventions. The namespace com.thinkrelevance consists of one or more parts separated by .. The actual name persistent is separated from the namespace by a /.

You have seen quite a few language features, but still no variables. Some things really do vary, and Chapter 6, *Concurrency*, on page 106 will show you how Clojure deals with changeable *references*. But most variables in traditional languages are unnecessary and downright dangerous. Let's see how Clojure gets rid of them.

2.7 Where's My For Loop?

Clojure has no for loop, and no direct mutable variables.[6] So how do you write all that code you are accustomed to writing with for loops?

Rather than create a hypothetical example, I decided to grab a piece of open-source Java code at (semi) random, find a method with some for loops and variables, and port it to Clojure. I opened the Apache Commons project, which is respected for quality and very widely used. I selected the StringUtils class in Commons Lang, assuming that such a class would require little domain knowledge to understand. I then browsed looking for a method that had multiple for loops and local variables, and found this:

Download exploring/StringUtils.java

```
// From Apache Commons Lang, http://commons.apache.org/lang/
public static int indexOfAny(String str, char[] searchChars) {
    if (isEmpty(str) || ArrayUtils.isEmpty(searchChars)) {
        return -1;
    }
    for (int i = 0; i < str.length(); i++) {
        char ch = str.charAt(i);
        for (int j = 0; j < searchChars.length; j++) {
            if (searchChars[j] == ch) {
                return i;
            }
        }
    }
    return -1;
}
```

6. Clojure provides *indirect* mutable references, but these must be explicitly called out in your code. See Chapter 6, *Concurrency*, on page 106 for details.

Metric	LOC	Branches	Exits	Variables
Imperative Version	14	4	3	3
Functional Version	2	0	1	0

Figure 2.3: Relative Complexity of Imperative and Functional indexOfAny

indexOfAny() walks str and reports the index of the first char that matches any char in searchChars, returning -1 if no match is found. Here are some example results from the documentation for indexOfAny():

```
StringUtils.indexOfAny(null, *)                = -1
StringUtils.indexOfAny("", *)                  = -1
StringUtils.indexOfAny(*, null)                = -1
StringUtils.indexOfAny(*, [])                  = -1
StringUtils.indexOfAny("zzabyycdxx",['z','a']) = 0
StringUtils.indexOfAny("zzabyycdxx",['b','y']) = 3
StringUtils.indexOfAny("aba", ['z'])           = -1
```

There are two ifs, two fors, three possible points of return, and three mutable local variables in indexOfAny(), and the method is fourteen lines long.[7]

Here is the Clojure port:

Download exploring/exploring.clj

```
(defn index-of-any [str chars]
  (some (fn [[idx char]] (and (get chars char) idx)) (indexed str)))
```

The Clojure version is simpler by every metric (See Figure 2.3). What accounts for the difference?

- The Java version begins with an if statement to check for empty/null collections. The Java version has to return -1 to indicate "Not found," since null is not a valid integer. Since Clojure can return null, the initial if is unncessary.
- The Java version uses a **for** loop to traverse the string. Clojure does not need a loop, as it can pass a function to some.

7. I generated the LOC using David A. Wheeler's SLOCCount (http://www.dwheeler.com/sloccount/). I personally feel a little silly counting the lines that are only closing braces, but that's the way it is generally done.

Prepared exclusively for Michael Kevin Hagel

- The Java version uses a variable i to track the position in the string. The Clojure version uses the indexed function, which returns a collection of pairs of the form [index, value].
- The Java version uses a variable ch for the current character in the String. The Clojure version calls an anonymous function that destructures its argument into the bindings it needs (idx and char), avoiding the need for a temporary let.
- The Java version uses a j to iterate through the set of characters. The Clojure version uses the set directly as a function to test whether the current character matches.

So it turns out that writing indexForAny() without for loops or variables is actually easier.[8] As a bonus, the code becomes shorter, easier to read, and less error prone. On larger units of code, these advantages become even more telling.

2.8 Finding Documentation

Clojure includes a number of functions that either provide documentation or let you reflect against the environment. The most basic of these is the doc() function, which returns the documentation for a function:

```
user=> (doc str)
-------------------------
clojure/str
([] [x] [x & ys])
  With no args, returns the empty string. With one arg x, returns
  x.toString().  (str nil) returns the empty string. With more than
  one arg, returns the concatenation of the str values of the args.
```

The first line of doc's output contains the fully-qualified name of the function. The next line contains the possible argument lists, generated directly from the code. Finally, the remaining lines contain the function's *doc-string*, if the function definition included one.

You can add a doc-string to your own functions by placing it immediately after the function name.

Download exploring/exploring.clj

```
(defn hello
```

8. It is worth mentioning that you could write a functional indexForAny() in plain Java, although it would not be idiomatic. It may become more idiomatic when closures are added to the langauge. See http://functionaljava.org/ for more information.

```
  "Writes hello message to *out*. Calls you by name"
  [name]
  (println (str "Hello, " name)))
```

Sometimes you will not know the exact name of what you are searching for. The find-doc() function will search for anything whose doc() output contains a string you pass in. Maybe you are wondering how Clojure does reduce:

```
user=> (find-doc "reduce")
-------------------------
clojure/areduce
([a idx ret init expr])
Macro
   ... details elided ...
-------------------------
clojure/reduce
([f coll] [f val coll])
   <ldots/> details elided ...
```

Interesting. In Clojure, reduce() seems to have a special case variant areduce() for arrays.

You can use methods such as class, and ancestors, and instance? methods to reflect against the underlying Java object model. You can tell, for example, that Clojure's collections are also Java collections:

```
user=> (ancestors (class [1 2 3]))
#{java.util.List clojure.lang.IPersistentVector
  java.lang.Object java.util.Comparator
  java.io.Serializable java.lang.Iterable
  java.util.Collection clojure.lang.APersistentVector
  java.util.RandomAccess clojure.lang.IObj clojure.lang.AFn
  java.lang.Comparable clojure.lang.Obj clojure.lang.IFn}
```

Clojure's complete API is documented online at http://clojure.org/api. The right sidebar links to all functions and macros by name, and the left sidebar links to a set of overview articles on various Clojure features. You can download a PDF version of the online documentation from the Clojure Google Group's file archive;[9] the file name is manual.pdf.

2.9 Wrapping Up

This has been a long chapter. But think how much ground you have covered: You can instantiate basic literal types, define and call functions, manage namespaces, and read and write metadata. You can write

9. http://groups.google.com/group/clojure/files

purely functional code, and yet you can easily introduce side effects when you need to. You have also met Lisp concepts including reader macros, special forms, and destructuring.

The material here would take hundreds of pages to cover in most other languages. Is the Clojure way really that much simpler? Yes, in part. Half the credit for this chapter belongs to Clojure. Clojure's elegant design and abstraction choices make the language much easier to learn than most.

That said, the language may not *seem* so easy to learn right now. That's because we are taking advantage of Clojure's power to move much faster than most programming language books.

So the other half of the credit for this chapter belongs to you, the reader. Clojure will give back what you put in, and then some. Take the time you need to feel comfortable with the chapter's examples, and with using the REPL.

In the next chapter, we will see how Clojure interoperates seamlessly with Java libraries.

Prepared exclusively for Michael Kevin Hagel

Chapter 3

Working with Java

Clojure's Java support is the both powerful and lean. Powerful, in that it bring all the expressiveness of Lisp syntax, plus some syntactic sugar tailored to Java. Lean, in that can get right to the metal. Clojure code compiles to bytecode, and does not have to go through any special translation layer on the way to Java.

Clojure embraces Java and its libraries. Idiomatic Clojure code calls Java libraries directly, and does not try to wrap everything under the Sun to look like Lisp. This surprises many new Clojure developers, but is very pragmatic. Where Java isn't broken, Clojure doesn't fix it.

In this chapter, you will see how Clojure access to Java is convenient, elegant, and fast:

1. Calling Java is simple and direct. Clojure provides syntax extensions for accessing anything you could reach from Java code: classes, instances, constructors, methods, and fields. While you will typically call Java code directly, you can also wrap Java APIs and use them in a more functional style.

2. Clojure is *fast*, unlike many other dynamic languages on the JVM. You can use custom support for primitives and arrays, plus type hints, to cause Clojure's compiler to generate the same code that a Java compiler would generate.

3. Java code can call Clojure code, too. Clojure can generate Java classes on the fly. On a one-off basis, you can use `proxy`, or you can generate and save classes with `gen-and-save-class`.

4. Clojure's exception handling is easy to use. Better yet, explicit exception handling is rarely necessary. Clojure's exception prim-

Prepared exclusively for Michael Kevin Hagel

itives are the same as Java's. However, Clojure does not require you to deal with checked exceptions, and makes it easy to cleanup resources with the with-open idiom.

3.1 Calling Java

Clojure provides simple, direct syntax for calling Java code: creating objects, invoking methods, and accessing statics methods and fields. In addition, Clojure provides syntactic sugar that makes calling Java from Clojure more concise than calling Java from Java!

Not all types in Java are created equal: the primitives and arrays work differently. Where Java has special cases, Clojure gives you direct access to these as well. Finally, Clojure provides a set of convenience functions for common tasks that would be unwieldy in Java.

Accessing Constructors, Methods, and Fields

The first step in many Java interop scenarios is creating a Java object. Clojure provides the

```
(new classname)
```

special form for this purpose. Try creating a new Random:

```
(new java.util.Random)
-> java.util.Random@4f1ada
```

The REPL simply prints out the new Random instance by calling its toString() method. In order to use a Random, you will need to save it away somewhere. For now, simply use def to save the Random into a Clojure Var:

```
(def rnd (new java.util.Random))
-> #'user/rnd
```

Now you can call methods on rnd using Clojure's . special form.

```
(. class-or-instance member-symbol & args)
(. class-or-instance (member-symbol & args))
```

The . can call methods. For example, the following code calls the no-argument version of nextInt():

```
(. rnd nextInt)
-> -791474443
```

Random also has a nextInt() that takes an argument. You can call that version by simply adding the argument to the list:

```
(. rnd nextInt 10)
-> 8
```

The . special form works with fields as well as with methods, and with statics as with instances. Here you can see the . used to get the value of pi:

```
(. Math PI)
-> 3.141592653589793
```

Notice that Math is not fully qualified. It doesn't have to be, because clojure imports java.lang automatically. To avoid typing java.util.Random everywhere, you could explicitly import it.

```
(import [& import-lists])
; import-list => (package-symbol & class-name-symbols)
```

import takes a variable number of lists, with the first part each list being a package name, and the rest being names to import from that package. The following import allows unqualified access to both Random and Locale.

```
(import '(java.util Random Locale))
-> nil

Random
-> java.util.Random

Locale
-> java.util.Locale
```

At this point, you have almost everything you need to call Java from Clojure. You can:

1. Import classnames
2. Create instances
3. Access fields
4. Invoke methods

However, there isn't anything particularly exciting about the syntax. It is just "Java with parentheses." In the next section, you will see how Clojure provides syntactic sugar to ease Java interop.

Syntactic Sugar

Most of the Java forms shown in the previous section have a shorter form. Instead of new, you can use the Classname. form. The following are equivalent:

```
(new Random)
(Random.)
```

For static methods and fields, the short form is Classname/membername. The following are equivalent:

```
(. Math PI)
Math/PI
```

Another short form is .methodOrFieldName, and it comes *first* in the form. The following calls are equivalent.

```
(. rnd nextInt)
(.nextInt rnd)
```

The Java APIs often introduce several layers of indirection between you and where you want to be. For example, to find out the URL for the source code of a particular object, you need to chain through the following objects: the class, the protection domain, the code source, and finally the location. The prefix-dot notation gets ugly fast:

```
(.getLocation
 (.getCodeSource (.getProtectionDomain (.getClass '(1 2)))))
-> file:/Users/stuart/repos/clojure/clojure.jar
```

Clojure's .. macro cleans this up.

```
(.. class-or-instance form & forms)
```

The .. chains together multiple member accesses by making each result the 'this' object for the next member access in the chain. Looking up an object's code url becomes:

```
(.. '(1 2) getClass getProtectionDomain getCodeSource getLocation)
-> file:/Users/stuart/repos/clojure/clojure.jar
```

The .. reads left-to-right, like Java, not inside-out, like Lisp. For longer expressions, it is shorter than the pure-Java equivalent in both characters and number of parentheses.

The .. macro is great if the result of each operation is an input to the next. Sometimes you don't care about the results of method calls, and simply want to make several calls on the same object. The doto macro makes it easy to make several calls on the same object:

```
(doto class-or-inst & member-access-forms)
```

As the "do" in doto suggests, you can use doto to cause side effects in the mutable Java world. For example, use doto to set multiple system properties:

```
(doto (System/getProperties)
  (setProperty "name" "Stuart")
  (setProperty "favoriteColor" "blue"))
```

Clojure's syntactic sugar makes code that calls Java shorter and easier-to-read. In idiomatic Clojure, prefer the (ClassName.), (..), and (ClassName/static) forms over the more generic (new ClassName) and (.) forms.

Using Java Collections

Clojure's collection classes supplant the Java collections for most purposes. Clojure's collections are concurrency-safe, have good performance characteristics, and implement the Java collection interfaces where appropriate. So, you should be able to use Clojure's collections from with Clojure, and even pass them back into Java if need be.

If you do choose to use the Java collections, nothing in Clojure will stop you. From Clojure's perspective, the Java collections are classes like any other, and all the various forms above will work. But you won't have the benefit of Clojure's function style.

The one place where you will need to deal with Java collections is the special case of Java arrays. In Java, arrays are have their own syntax, and their own bytecode instructions. Java arrays do not implement any Java interface. Clojure collections cannot masquerade as arrays. (Java collections can't either!) The Java platform makes arrays a special case in every way, so Clojure does too.

Clojure provides make-array to create Java arrays.

```
(make-array class length)
(make-array class dim & more-dims)
```

make-array takes a class and a variable number of array dimensions. For a one-dimensional array of Strings you might say:

```
(make-array String 5)
-> [Ljava.lang.String;@45a270b2
```

The odd output is courtesy of Java's implementation of toString()for arrays: [Ljava.lang.String is the JVM Specification's encoding for "one-dimensional array of Strings." That's not very useful at the REPL, so you can use Clojure's seq to wrap any Java array as a Clojure sequence, so that the REPL can print the individual array entries:

```
(seq (make-array String 5))
-> (nil nil nil nil nil)
```

Prepared exclusively for Michael Kevin Hagel

Clojure provides a set of low-level operations on Java arrays, including aset, aget, alength.

```
(aset java-array index value)
(aset java-array index-dim1 index-dim2 ... value)

(aget java-array index)
(aget java-array index-dim1 index-dim2 ...)

(alength java-array)
```

Use make-array to create an array, and then experiment with using aset, aget, and alength to work with the array:

```
(defn painstakingly-create-array []
  (let [arr (make-array String 5)]
    (aset arr 0 "Painstaking")
    (aset arr 1 "to")
    (aset arr 2 "fill")
    (aset arr 3 "in")
    (aset arr 4 "arrays")
    arr))

(aget (painstakingly-create-array) 0)
-> "Painstaking"

(alength (painstakingly-create-array))
-> 5
```

Most of the time, you will find it simpler to use higher-level functions such as to-array, which creates an array directly from any sequence.

```
(to-array sequence)
```

to-array always creates an Object array:

```
(to-array ["Easier" "array" "creation"])
-> #<[Ljava.lang.Object;@1639f9e3>
```

to-array's cousin into-array creates an array with a more specific type based on the first object you pass to it.

```
(into-array type seq)
(into-array seq)
```

You can pass an explicit type as an optional first argument to into-array:

```
(into-array String ["Easier", "array", "creation"])
-> #<[Ljava.lang.String;@391ecf28>
```

If you omit the type argument, into-array will guess the type based on the first item in the sequence.

```
(into-array ["Easier" "array" "creation"])
-> #<[Ljava.lang.String;@76bfd849>
```

Prepared exclusively for Michael Kevin Hagel

Java's array toString() is not very informative, but you can see that the array contains Strings, not Objects.

Convenience Functions

Clojure provides several handy convenience functions for working with Java code. For example, consider the mismatch between Clojure functions and Java methods. Clojure functions are often passed to other functions. If you want to triple all the items in a collection, pass triple to map:

```
(defn triple [x] (* x 3))
-> #'user/triple

(map triple (range 1 11))
-> (3 6 9 12 15 18 21 24 27 30)
```

triple is such a simple function that it probably isn't worth writing. You could just inline it with a function literal:

```
(map (fn[x] (* x 3)) (range 1 11))
-> (3 6 9 12 15 18 21 24 27 30)
```

Or even the uber-terse

```
(map #(* % 3) (range 1 11))
-> (3 6 9 12 15 18 21 24 27 30)
```

Now let's make some strings bigger by upcasing them. Since toUpperCase() is a method on Java strings, and since Clojure strings *are* Java strings, you might want this to work:

```
(map .toUpperCase ["a" "short" "message"])
-> java.lang.Exception:\
   Unable to resolve symbol: .toUpperCase in this context
```

The problem is that toUpperCase() is a Java method, not a Clojure function. You could write your own function that turns around and invokes toUpperCase():

```
(map (fn [x] (.toUpperCase x)) ["a" "short" "message"])
-> ("A" "SHORT" "MESSAGE")
```

The method-as-function idiom is a common one, so Clojure provides the memfn macro to wrap methods for you:

```
(map (memfn toUpperCase) ["a" "short" "message"])
-> ("A" "SHORT" "MESSAGE")
```

Another common idiom is checking whether an object is an instance of a certain class. Clojure provides the instance? function for this purpose:

Prepared exclusively for Michael Kevin Hagel

```
(instance? Integer 10)
-> true

(instance? Comparable 10)
-> true

(instance? String 10)
-> false
```

If you need to read data from existing Java beans, you can convert them to Clojure maps for convenience. Use Clojure's bean function to wrap a Java bean in an immutable Clojure map.

```
(bean java-bean)
```

For example, the following code returns the properties of a Cipher instance:

```
(import '(javax.crypto Cipher))
-> nil

(bean (Cipher/getInstance "DES/CBC/PKCS5Padding"))
-> {:IV nil, :algorithm "DES/CBC/PKCS5Padding",
   :parameters nil, :blockSize 8, :exemptionMechanism nil,
   :provider SunJCE version 1.6, :class javax.crypto.Cipher}
```

Once you have converted a Java bean into a Clojure map, you can use any of Clojure's map functions. For example, you can use a symbol as a function to extract a particular field:

```
(:blockSize (bean (Cipher/getInstance "DES/CBC/PKCS5Padding")))
-> 8
```

Beans have never been easier to use.

3.2 Optimizing for Performance

In Clojure, it is idiomatic to call Java using the techniques described in Section 3.1, *Calling Java*, on page 58. The resulting code will be fast enough for ninety percent of scenarios. When you need to, though, you can make localized changes to boost performance. These changes will not change how outside callers invoke your code, so you are free to make your code work, *then* make it fast.

Using Primitives for Performance

In the preceding sections, function parameters carry no type information. Clojure simply does the right thing. Depending on your perspective, this is either a strength or a weakness. A strength, because your code is clean and simple, and can take advantage of duck typing. But

Why Duck Typing?

With duck typing, an object's type is the sum of *what it can do* (methods), rather than the sum of *what it is* (the inheritance hierarchy). In other words, it is more import to quack() and fly() than it is to insist that you implement the Duck interface.

Duck typing has two major benefits:

- Duck-typed code is easier to test. Since the rules of "what can go here" are relaxed, it is easier to isolate code under test from irrelevant dependencies.
- Duck-typed code is easier to reuse. Reuse is more granular, at the method level instead of the interface level. So it is more likely that new objects can be plugged in directly, without refactoring or wrapping.

If you think about it, the "easier to test" argument is just a special case of the "easier to reuse" argument. Using code inside a test harness is a kind of reuse. (Testing should generally be the *first* use a piece of code sees, but I will save the Test-Driven-Development (TDD) rant for another day.)

Idiomatic Clojure takes advantage of duck typing, but you can add type hints Section 3.2, *Adding Type Hints*, on page 68 for performance or documentation purposes.

also a weakness, because a reader of the code cannot be certain of data types, and because doing the right thing carries some performance overhead.

Consider a function that calculates the sum of the numbers from 1 to n:

Download interop/interop.clj

```
(defn sum-to [n]
  (loop [i 1 sum 0]
    (if (<= i n)
      (recur (inc i) (+ i sum))
      sum)))
```

You can verify that this function works with a small input value:

```
(sum-to 10)
-> 55
```

Let's see how sum-to performs. To time an operation, you can use the time. When benchmarking, you tend to want to take several measurements so that you can eliminated startup overhead plus any outliers, so you can wrap time in a dotimes:

```
user=> (dotimes _ 5 (time (sum-to 10000)))
"Elapsed time: 4.203 msecs"
"Elapsed time: 2.495 msecs"
"Elapsed time: 2.077 msecs"
"Elapsed time: 1.832 msecs"
"Elapsed time: 1.417 msecs"
```

To speed things up, you can ask Clojure to treat n, i, and sum as ints:

Download interop/interop.clj

```
(defn integer-sum-to [n]
  (let [n (int n)]
    (loop [i (int 0) sum (int 0)]
      (if (<= i n)
        (recur (inc i) (+ i sum))
        sum))))
```

The integer-sum-to is indeed faster, by more than an order of magnitude:

```
(dotimes _ 5 (time (faster-sum-to 10000)))
"Elapsed time: 0.129 msecs"
"Elapsed time: 0.09 msecs"
"Elapsed time: 0.09 msecs"
"Elapsed time: 0.09 msecs"
"Elapsed time: 0.09 msecs"
```

Clojure's convenient math operators (+, -, etc.) make sure their arguments do not overflow. Maybe you can get an even faster function by using the unchecked version of +, unchecked-add:

Download interop/interop.clj

```
(defn unchecked-sum-to [n]
  (let [n (int n)]
    (loop [i (int 0) sum (int 0)]
      (if (<= i n)
        (recur (inc i) (unchecked-add i sum))
        sum))))
```

The unchecked-sum-to is not enough faster than integer-sum-to to merit the risk of undetected overflow:

```
(dotimes _ 5 (time (unchecked-sum-to 10000)))
"Elapsed time: 0.779 msecs"
"Elapsed time: 0.052 msecs"
"Elapsed time: 0.051 msecs"
"Elapsed time: 0.052 msecs"
```

Prepared exclusively for Michael Kevin Hagel

```
"Elapsed time: 0.052 msecs"
```

Both integer-sum-to and unchecked-sum-to lead to bad outcomes if the calculation overflows:

```
; accurate but slow
(sum-to 100000)
-> 5000050000

; visibly fails
(integer-sum-to 100000)
-> java.lang.ArithmeticException: integer overflow

; wrong answer!!
(unchecked-sum-to 100000)
-> 705082704
```

Given the competing concerns of correctness and performance, you should normally prefer a simple, undecorated loop such as the original sum-to. If profiling identifies a bottleneck, you can force Clojure to use a primitive type in just the places that need it.

The example of summing the numbers from 1 to n demonstrates a general rule for Clojure: Whenever you find yourself about to use a loop, check to see if there is a built-in function that already performs the same loop. Summing a sequence is the same as summing the first two items, adding that result to the next item, and so on. That is exactly the loop that reduce provides. With reduce, you can rewrite sum-to as a one-liner:

```
(defn better-sum-to [n]
  (reduce + (range 1 (inc n))))
```

The example also demonstrates an even more general point: *Pick the right algorithm to begin with.* The sum of numbers from 1 to n can be calculated directly as follows.

```
(defn best-sum-to [n]
  (/ (* n (inc n)) 2))
```

Even without performance hints, this is faster than implementations based on repeated addition:

```
(dotimes _ 5 (time (best-sum-to 10000)))
"Elapsed time: 0.037 msecs"
"Elapsed time: 0.018 msecs"
"Elapsed time: 0.0050 msecs"
"Elapsed time: 0.0040 msecs"
"Elapsed time: 0.0050 msecs"
```

Prepared exclusively for Michael Kevin Hagel

Performance is a tricky subject. Don't write ugly code in search of speed. Start by choosing appropriate algorithms, and getting your code to simply work correctly. If you have performance issues, profile to identify the problems. Then, introduce only as much complexity as you need to solve those problems.

Adding Type Hints

Clojure supports adding type hints to function parameters, let bindings (see Section 2.4, *Bindings and Namespaces*, on page 42), variable names, and expressions. These type hints serve two purposes:

- Optimize critical performance paths.
- Document the required type, and enforce it at runtime.

For example, consider the following function, which returns information about a Java class:

Download interop/interop.clj

```
(defn describe-class [c]
  {:name (.getName c)
   :final (java.lang.reflect.Modifier/isFinal (.getModifiers c))})
```

You can ask Clojure how much type information it can infer, by setting *warn-on-reflection* to true:

```
(set! *warn-on-reflection* true)
-> true
```

With *warn-on-reflection* set to true, compilation of describe-class will produce the following warnings:

```
Reflection warning, line: 87 - reference to field getName can't be resolved.
Reflection warning, line: 88 - reference to field getModifiers can't be resolved.
```

These warnings indicate that Clojure has no way to know the type of c. You can provide a type hint to fix this, using the metadata syntax #^ClassName:

Download interop/interop.clj

```
(defn describe-class [#^Class c]
  {:name (.getName c)
   :final (java.lang.reflect.Modifier/isFinal (.getModifiers c))})
```

With the type hint in place, the reflection warnings will disappear. The compiled Clojure code will be exactly the same as compiled Java code. Further, attempts to call describe-class with something other than a Class will fail with a ClassCastException:

```
user=> (describe-class StringBuffer)
{:name "java.lang.StringBuffer", :final true}

user=> (describe-class "foo")
java.lang.ClassCastException: \
java.lang.String cannot be cast to java.lang.Class
```

When you need speed, type hints will let Clojure code compile down to the same code Java will produce. But you won't need type hints that often. Make your code *right* first, then worry about making it fast.

3.3 Calling Clojure from Java

Clojure's objects all implement reasonable Java interfaces:

- Clojure's data structures implement interfaces from the Java Collections API.
- Clojure's functions implement Runnable and Callable

In addition to these generic interfaces, you will occasionally need domain-specific interfaces. Often this comes in the form of callback handlers for event driven APIs such as Swing or some XML parsers. Clojure can generate one-off proxies or classes when needed, using a fraction of the lines of code necessary in Java.

Creating Java Proxies

In order to interoperate with Java, you will often need to implement Java interfaces. Clojure makes this easy, using the ASM library[1] to generate Java bytecodes when you need them.

A good example is parsing XML with a Simple API for XML (SAX) parser. To get ready for this example, go ahead and import the following classes (we'll need them all before we are done:

```
(import '(org.xml.sax InputSource)
  '(org.xml.sax.helpers DefaultHandler)
  '(java.io StringReader)
  '(javax.xml.parsers SAXParserFactory))
```

To use a SAX parser, you need to implement a callback mechanism. The easiest was is often to extend the DefaultHandler class. In Clojure, you can do this with the proxy function

```
(proxy class-and-interfaces super-cons-args & fns)
```

1. http://asm.objectweb.org/

As a simple example, use proxy to create a DefaultHandler that prints the details of all calls to startElement:

Download interop/interop.clj

```
(def print-element-handler
     (proxy [DefaultHandler] []
       (startElement
        [name, local, qname, atts]
        (println (format "Saw element: %s" qname)))))
```

proxy generates an instance of a proxy class. The first argument to proxy is [DefaultHandler], a vector of the superclass and superinterfaces. The second argument [], is a vector of arguments to the base class constructor. In this case none are needed

After the proxy setup comes the implementation code for zero or more proxy methods. The proxy above has one method. Its name is startElement, and it takes four arguments and prints the name of the name arg.

Now all you need is a parser to pass the handler to. This requires plowing through a pile of Java factory methods and constructors. For a simple exploration at the REPL, you can create a function that parses XML in a String:

```
(defn demo-sax-parse [source handler]
  (.. SAXParserFactory newInstance newSAXParser
      (parse (InputSource. (StringReader. source))
       handler)))
```

Now the parse is easy.

```
(demo-sax-parse "<foo>
<bar>Body of bar</bar>
</foo>" print-element-handler)
Saw element: foo
Saw element: bar
```

The example above demonstrates the mechanics of creating a Clojure proxy to deal with Java's XML interfaces. You can take a similar approach to implementing your own custom Java interfaces. But if all you are doing is XML processing, Clojure-Contrib already has terrific XML support, and can work with any SAX-compatible Java parser. See the (as yet) unwritten *sec.clojure.contrib.lazy-xml* for details.

The proxy mechanism is completely general, and can be used to generate any kind of Java object you want, on the fly. Sometimes the objects are so simple you can fit the entire object in a single line. The following code

Prepared exclusively for Michael Kevin Hagel

creates a new thread, then creates a new dynamic subclass of Runnable to run on the new thread:

```
(.start (Thread.
          (proxy [Runnable] [] (run [] (println "I ran!")))))
```

In Java, you must provide an implementation of method on every interface you **implement**. In Clojure, you can leave them out:

```
(proxy [Callable] []) ; proxy with no methods (??)
```

If you omit a method implementation, Clojure provides a default implementations that throws an UnsupportedOperationException:

```
(.call (proxy [Callable] []))
-> java.lang.UnsupportedOperationException: call
```

The default implementation does not make much sense for interfaces with only one method, such as Runnable and Callable, but can be handy when you are implementing larger interfaces and don't care about some of the methods.

So far in this section, you have seen how to use proxy to create implementations of Java interfaces. This is very powerful when you need it, but often Clojure is already there on your behalf. For example, functions automatically implement Runnable and Callable:

```
; normal usage: call an anonymous function
(#(println "foo"))
-> foo

; call through Runnable's run
(.run #(println "foo"))
-> foo

; call through Callable's call
(.call #(println "foo"))
-> foo
```

This makes is very easy to pass Clojure functions to other threads:

Download interop/interop.clj

```
(dotimes i 5
  (.start
   (Thread.
    (#(fn []
        (Thread/sleep (rand 1000))
        (println "Finished %d on %s" i (Thread/currentThread)))))))
```

For one-off tasks likes XML and thread callbacks, Clojure's proxies are quick and easy to use. If you need a longer-lived class, you can generate new named classes from Clojure as well.

Creating Java Classes

Clojure can create named Java classes that can plug directly into existing Java architectures. Here's an example scenario: You have a large investment in an existing Java codebase, including continuous integration around a suite of JUnit tests. You want to write your new Widget subsystem in Clojure, and you want the tests for Widget to run as part of your existing JUnit test suite.

Since JUnit is not a part of Java proper, you will need to add it to the classpath before you try to generate a JUnit test class. You can do this by launching Clojure with junit.jar added to the classpath. If you are already in the middle of a Clojure session, no problem. Just use

```
(add-classpath url)
```

The following call to add-classpath adds JUnit to the classpath.[2]

```
(add-classpath "file:///Users/stuart/devtools/java/junit-4.4.jar")
-> nil
```

Now you are ready to create your test class. You can use Clojure's gen-and-save-class to create a Java class and save the bytes to a file.

```
(gen-and-save-class path name & options)
```

The first two arguments are required: a path to save the generated class, and a class name. The options can specify a base class, superinterfaces, constructors, methods, factory functions and accessors. Here is how you would call gen-and-save-class to create a WidgetTest with two test methods testOne() and testTwo(). (Note that gen-and-save-class will follow Java's recommended directory structure, so you will need to create a test directory to receive the generated class.)

Download interop/interop.clj

```
(gen-and-save-class
 "." "test.WidgetTest"
 :extends junit.framework.TestCase
 :methods [
           ["testOne" [] (Void/TYPE)]
           ["testTwo" [] (Void/TYPE)]])
```

2. If this exact url works for you, you have my laptop. Please give it back.

The interesting piece here is the vector of method data. Each method is represented by a name, argument list, and return value. The empty argument list and void return value make these methods conform to JUnit's notion of a test method.

If the call succeeds, nothing much happens in the REPL. But you should find a test/WidgetTest.class file on your filesystem.

The next step is to try running the JUnit test. You will need Clojure, JUnit, and the generated WidgetTest.class file on your classpath. On my system, I have created a CLOJURE_CLASSPATH environment variable that includes all these paths, so the command will look like this:

```
code$ java -cp $CLOJURE_CLASSPATH junit.textui.TestRunner test.WidgetTest
.E.E
Time: 0.001
There were 2 errors:
1) testOne(test.WidgetTest)java.lang.UnsupportedOperationException:\
   testOne (test/WidgetTest-testOne not defined?)
```

WidgetTest is complaining that the test methods are not defined. This is true. In the call to gen-and-save-class you specified the method signatures, but no method bodies. Clojure uses a naming convention to locate the method bodies. For test/WidgetTest.class, Clojure will look for the file test/WidgetTest.clj. Inside the file, Clojure will expect to find function names following the convention WidgetTest-testOne, in a namespace that matches the Java package:

Download test/WidgetTest.clj

```
(ns test
    (import (junit.framework Assert)))

(defn WidgetTest-testOne [_]
  (Assert/assertEquals 1 1))

(defn WidgetTest-testTwo [_]
  (Assert/fail "fix me"))
```

Now you should be able to run WidgetTest from the command line:

```
code$ java -cp $CLOJURE_CLASSPATH junit.textui.TestRunner test.WidgetTest
..F
Time: 0.002
There was 1 failure:
1) testTwo(test.WidgetTest)junit.framework.AssertionFailedError: fix me
```

The gen-and-save-class function delegates to another function, gen-class, which simply creates the classfile bytes. These low-level tools give you the flexibility to give your Clojure app any Java face it needs.

Prepared exclusively for Michael Kevin Hagel

Loading Clojure into a Running Java Application

3.4 Exception Handling

In Java code, exception handling crops up for three reasons:

- Wrapping checked exceptions.
- Cleaning up non-memory resources such as file and network handles
- Responding to the problem: ignoring the exception, retrying the operation, converting the exception to a non-exceptional result, etc.

In Clojure, things are similar, but simpler. The try and throw special forms give you all the capabilities of Java's **try**, **catch**, **finally**, **throw**. But you should not have to use them very often, because:

- You do not have to deal with checked exceptions in Clojure.
- You can use macros like with-open to encapsulate resource cleanup.

Let's see what this looks like in practice.

Wrapping Checked Exceptions

Java programs often wrap checked exceptions at abstraction boundaries. A good example is Apache Ant, which tends to wrap low-level exceptions (such as I/O exceptions) with an Ant-level build exception:

```
// Ant-like code (simplified for clarity)
try {
  newManifest = new Manifest(r);
} catch (IOException e) {
  throw new BuildException(...);
}
```

In Clojure, you are not forced to deal with checked exceptions. You do not have to catch them, or declare that you throw them. So, the code above would translate to:

```
(.Manifest r)
```

The absence of exception wrappers makes idiomatic Clojure code easier to read, write, and maintain than idiomatic Java. (That said, nothing prevents you from wrapping exceptions in Clojure. It simply is not required.

Prepared exclusively for Michael Kevin Hagel

Cleaning Up Resources

Garbage collection will clean up resources in memory. If you use resources that live outside of garbage-collected memory, such as file handles, you need to make sure that you clean them up, even in the event of an exception. In Java, this is normally handled in a finally block.

If the resource you need to free follows the convention of having a close method, you can use Clojure's with-open macro.

```
(with-open name init-form & body)
```

Internally, with-open creates a try block, sets name to the result of init-form, and then runs the forms in body. Most importantly, with-open always closes the object bound to name in a finally block.

If you need to do something other than close in a finally block, the Clojure try form looks like this:

```
(try expr* catch-clause* finally-clause?)
catch-clause -> (catch classname name expr*)
finally-clause -> (finally expr*)
```

It can be used thusly:

```
(try
 (throw (Exception. "something failed"))
 (finally
  (println "we get to clean up")))
-> we get to clean up
java.lang.Exception: something failed
```

The fragment above also demonstrates Clojure's throw form, which simply throws whatever exception is passed to it.

Responding to an Exception

The most interesting case is when an exception handler attempts to respond to the problem in a catch block. As a simple example, consider writing a function to test whether a particular class is available at runtime:

Download interop/interop.clj

```
; not caller-friendly
(defn class-available? [class-name]
  (Class/forName class-name))
```

This approach is not very caller-friendly. The caller simply wants a yes/no answer, but instead gets an exception:

```
(class-available? "borg.util.Assimilate")
-> java.lang.ClassNotFoundException: borg.util.Assimilate
```

A friendlier approach uses a catch block to return false:

Download interop/interop.clj

```
(defn class-available? [class-name]
  (try
   (Class/forName class-name) true
   (catch ClassNotFoundException _ false)))
```

The caller experience is much better now:

```
(class-available? "borg.util.Assimilate")
-> false

(class-available? "java.lang.String")
-> true
```

Clojure gives you everything you need to throw and catch exceptions, and to cleanly release resources. At the same time, Clojure keeps exceptions in their place. They are important, but not so important that your mainline code is dominated by the exceptional.

3.5 Wrapping Up

Clojure code can call directly into Java, and can implement Java classes and interfaces where necessary. Do not be afraid to drop to Java when you need it. Clojure is pragmatic, and does not aspire to wrap or replace Java code that already works.

One part of Java that you will use rarely is the Collections API. Clojure provides a powerful, functional, thread-safe alternative to Java collections: the sequence library, In the next chapter, you will meet Clojure's ubiquitous sequences.

Prepared exclusively for Michael Kevin Hagel

Chapter 4

Unifying Data with Sequences

Programs manipulate data. At the lowest level, programs work with structures such as strings, lists, vectors, maps, sets, and trees. At a higher level, these same data structure abstractions crop up again and again. For example:

- XML data is tree
- Database result sets can be viewed as lists or vectors
- Directory hierarchies are trees
- Files are often viewed as one big string, or as a vector of lines

In Clojure, all these data structures are unified under a single abstraction: the sequence (seq). A seq (pronounced "seek") is a *logical* list. Logical, because Clojure does not tie sequences to *implementation details* of a list such as a Lisp cons cell. Instead, the seq is an abstraction that can be used everywhere.

Objects that can be viewed as seqs are called seq-able (pronounced 'seek-a-bull). In this chapter you will meet a variety of seq-able things:

- all the Clojure collections
- all Java collections (and arrays)
- regular expression matches
- directory structures
- streams
- XML trees
- SQL results

Prepared exclusively for Michael Kevin Hagel

You will also meet the sequence library, a set of functions that can work with any seq-able. Because so many things are sequences, the sequence library is much more powerful and general than the collection APIs in most languages. The sequence library includes functions to create, filter, and transform data. These functions act as the collections API for Clojure, and they also replace many of the loops you would write in an imperative language.

In this chapter you will become a power user of Clojure sequences. You will see how to use a common set of very expressive functions with an incredibly wide range of data types. Then, in the next chapter, Chapter 5, *Functional Programming*, on page 105, you will learn the functional style in which the sequence library is written.

4.1 Everything is a Sequence

Every aggregate data structure in Clojure can be viewed as a sequence. A sequence has three core capabilities:

- You can get the first item in a sequence.

  ```
  (first seq)
  ```

- You can get everything but the first item, a.k.a. the rest of a sequence:

  ```
  (rest seq)
  ```

- You can add a new item to the front of the sequence. For historical reasons, this is called consing:

  ```
  (cons elem seq)
  ```

Under the hood, these three capabilities are declared in a Java interface clojure.lang.ISeq. (Keep this in mind when reading about Clojure, because the name ISeq is often used interchangeably with seq.)

If you have a Lisp background, you expect to find that the seq functions work for lists:

```
(first '(1 2 3))
-> 1
(rest '(1 2 3))
-> (2 3)
(cons 0 '(1 2 3))
-> (0 1 2 3)
```

In Clojure, the same functions will work for other data structures as well. You can treat vectors as seqs:

The Origin of Cons

Clojure's sequence is an abstraction based on Lisp's concrete lists. In the original implementation of Lisp, the three fundamental list operations were named car, cdr, and cons. car and cdr are acronyms that refer to implementation details of Lisp on original IBM 704 platform. Many Lisps, including Clojure, replace these esoteric names with the more meaningful names first and rest.

The third function, cons, is short for construct. Lisp programmers use cons as a noun, verb, and adjective. You can cons a cons cell, which is called a cons for short.

Most Lisps, including Clojure, retain the original cons name, since "construct" is a pretty good mnemonic for what cons does. It also helps to remind you that sequences are immutable. For convenience you might say that cons adds an element to a sequence, but it is more accurate to say that cons*constructs* a new sequence, which is like the original sequence but with one element added.

```
(first [1 2 3])
-> 1

(rest [1 2 3])
-> (2 3)

(cons 0 [1 2 3])
-> (0 1 2 3)
```

You can treat maps as seqs, if you think of a key/value pair as an item in the sequence.

```
(first {:fname "Stu" :lname "Halloway"})
-> [:lname "Halloway"]

(rest {:fname "Stu" :lname "Halloway"})
-> ([:fname "Stu"])

(cons [:mname "Dabbs"] {:fname "Stu" :lname "Halloway"})
-> ([:mname "Dabbs"] [:lname "Halloway"] [:fname "Stu"])
```

You can also treat sets as seqs.

```
(first #{:the :quick :brown :fox})
-> :brown
```

```
(rest #{:the :quick :brown :fox})
-> (:the :fox :quick)

(cons :jumped #{:the :quick :brown :fox})
-> (:jumped :brown :the :fox :quick)
```

Maps and sets have a stable traversal order, but that order depends on implementation detail and you should not rely on it. Elements of a set will not necessarily come back in the order that you put them in:

```
#{:the :quick :brown :fox}
-> #{:fox :the :brown :quick}
```

If you want a reliable order, you can use

```
(sorted-set & elements)
```

sorted-set will sort the values by their natural order:

```
(sorted-set :the :quick :brown :fox)
-> #{:brown :fox :quick :the}
```

Likewise, key/value pairs in maps won't necessarily come back in the order you put them in:

```
{:a 1 :b 2 :c 3}
-> {:a 1, :c 3, :b 2}
```

You can create a sorted map with

```
(sorted-map & elements)
```

sorted-maps won't come back in the order you put them in either, but they *will* come back sorted by key:

```
user=> (sorted-map :c 3 :b 2 :a 1)
{:a 1, :b 2, :c 3}
```

In addition to the core capabilities of seq, there are two other capabilities worth meeting immediately: conj and into.

```
(conj sequence element & elements)

(conj to-sequence from-sequence)
```

conj adds a single item to a collection, and into adds all the items in one collection to another. Both conj and into add items at an efficient insertion spot for the underlying data structure. For lists, conj and into add to the front:

```
(conj '(1 2 3) :a)
-> (:a 1 2 3)
```

Joe Asks...

Why do functions on vectors return lists?

When you try examples at the REPL, the results of rest and cons appear to be lists, even when the inputs are vectors, maps, or sets. Does this mean that Clojure is converting everything to a list internally? No! The sequence functions always return a seq, regardless of their inputs. You can verify this by checking the Java type of the returned objects:

```
(class '(1 2 3))
-> clojure.lang.PersistentList

(class (rest [1 2 3]))
-> clojure.lang.APersistentVector$Seq
```

As you can see, the result of (rest [1 2 3]) is some kind of Seq, not a List. So why does the result appear to be a list?

The answer lies in the REPL. When you ask the REPL to display a sequence, all it knows is that is has a sequence. It does not know what kind of collection the sequence was built from. So the REPL prints all sequences the same way: it walks the entire sequence, printing it as a list. (This distinction will be important in Section 4.3, *Lazy and Infinite Sequences*, on page 90, when we are dealing with lazy infinite sequences. Don't try to display an infinite sequence in the REPL!)

```
(into '(1 2 3) '(:a :b :c))
-> (:c :b :a 1 2 3)
```

For vectors, conj and into add elements to the back:

```
(conj [1 2 3] :a)
-> [1 2 3 :a]

(into [1 2 3] [:a :b :c])
-> [1 2 3 :a :b :c]
```

Because conj (and related functions) do the efficient thing for the underlying data structure, you can often write code that is both efficient and completely decoupled from a specific underlying implementation.

The Clojure sequence library is particularly suited for large (or even infinite) sequences. Most Clojure sequences are *lazy*: they generate ele-

ments only when they are actually needed. Thus, Clojure's sequence functions can process sequences too large to fit in memory.

Clojure sequences are *immutable*: they never change. This makes it easier to reason about programs, and means that Clojure sequences are safe for concurrent access. It does, however, create a small problem for human language. English-language descriptions flow much more smoothly when describing mutable things. Consider the two descriptions below, for a hypothetical sequence function triple.

- triple triples each element of a sequence.
- triple takes a sequence, and returns a new sequence with each element of the original sequence tripled.

The latter version is specific and accurate. The former is much easier to read, but it might lead to the mistaken impression that a sequence is actually changing. Don't be fooled: *Sequences never change.*. If you see the phrase "foo changes xml," mentally substitute "foo returns a changed copy of x."

4.2 Using the Sequence Library

The Clojure sequence library provides a rich set of functionality that can work with any sequence. If you come from an object-oriented background, the sequence library is truly "Revenge of the Verbs." The functions provide a rich backbone of functionality that can take advantage of any data structure that obeys the basic first/rest/cons contract.

The functions below are grouped into three broad categories:

- functions that create sequences
- functions that filter sequences
- sequence predicates
- functions that transform sequences

These divisions are somewhat arbitrary. Since sequences are immutable, *most* of the sequence functions create new sequences. Some of the sequence functions both filter and transform. Nevertheless, these divisions provide a rough road map through a large library.

Prepared exclusively for Michael Kevin Hagel

Creating Sequences

In addition to the sequence literals, Clojure provides a number of functions that create sequences. range produces a sequence from a start to an end, incrementing by step each time.

```
(range end)
(range start end)
(range start end step)
```

If not specified start defaults to zero and step defaults to 1. Try creating some ranges at the REPL:

```
(range 10)
-> (0 1 2 3 4 5 6 7 8 9)

(range 10 20)
-> (10 11 12 13 14 15 16 17 18 19)

(range 1 25 2)
-> (1 3 5 7 9 11 13 15 17 19 21 23)
```

The replicate function replicates an element xn times.

```
(replicate n x)
```

Try to replicate some items from the REPL:

```
(replicate 5 1)
-> (1 1 1 1 1)

(replicate 10 "x")
-> ("x" "x" "x" "x" "x" "x" "x" "x" "x" "x")
```

Both range and replicate represent ideas that can be extended infinitely. You can think of iterate as the infinite extension of range:

```
(iterate f x)
```

iterate begins with a value x, and continues forever, applying a function f to each value to calculate the next.

If you begin with 1, and iterate with inc you can generate the whole numbers:

```
(take 10 (iterate inc 1))
-> (1 2 3 4 5 6 7 8 9 10)
```

Since the sequence is infinite, you need another new function to help you view the sequence from the REPL.

```
(take n sequence)
```

take takes the first n items from a collection. You will see more about infinite sequences in Section 4.3, *Lazy and Infinite Sequences*, on page 90. For now, just remember not to look at them directly in the REPL. Always take just a few at a time.

The whole numbers are a pretty useful sequence to have around, so let's def them for future use:

```
(def whole-numbers (iterate inc 1))
-> #'user/whole-numbers
```

Just as iterate behaves like an infinite range, repeat behaves like an infinite replicate:

```
(repeat x)
```

Try repeating some elements at the REPL. Don't forget to wrap the result in a take:

```
(take 20 (repeat 1))
-> (1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)
```

The cycle function takes a collection, and cycles it infinitely:

```
(cycle coll)
```

Try cycling some collections at the REPL:

```
(take 10 (cycle (range 3)))
-> (0 1 2 0 1 2 0 1 2 0)
```

The interleave function takes multiple collections, and produces a new collection that interleaves values from each collection until one of the collections is exhausted.

```
(interleave & colls)
```

When one of the collections is exhausted, the interleave stops. So you can mix finite and infinite collections:

```
(interleave whole-numbers ["A" "B" "C" "D" "E"])
-> (1 "A" 2 "B" 3 "C" 4 "D" 5 "E")
```

Closely related to interleave is interpose, which returns a sequence with each of the elements of the input collection separated by a separator. You can use

```
(interpose separator coll)
```

to build delimited strings:

```
(interpose "," ["apples" "bananas" "grapes"])
-> ("apples" "," "bananas" "," "grapes")
```

interpose works nicely with (apply str ...) to produce output strings:

```
(apply str (interpose \, ["apples" "bananas" "grapes"]))
-> "apples,bananas,grapes"
```

The (apply str ...) idiom is common enough that clojure-contrib wraps it as

```
(str-join separator sequence)
```

Use str-join to comma-delimit a list of words:

```
(use 'clojure.contrib.str-utils)
-> nil

(str-join \, ["apples" "bananas" "grapes"])
-> "apples,bananas,grapes"
```

For each collection type in Clojure, there is a function that converts a list of arguments to that type:

```
(list & elements)

(vector & elements)

(set elements)

(hash-map key-1 val-1 ...)
```

Notice that set is a little different from the others. set expects a collection as its first argument:

```
(set [1 2 3])
-> #{1 2 3}
```

The other collection functions take a variable list of arguments

```
  (vector 1 2 3)
  -> [1 2 3]
  <code>

  <p>Now that you have the basics of creating sequences, you can use other Clojure ...*TRUNC*
</sect2>

<sect2 id="sec.filteringSequences">
  <title>Filtering Sequences</title>

  <p>Clojure provides a number of functions that filter a sequence, returning a sub...*TRUNC*

  <code language="clj" cite="filter">
  (filter pred coll)
```

filter takes a predicate and a collection, and returns a sequence of objects for which the filter returns true (when interpreted in a Boolean context).

You can filter the whole-numbers from the previous section to get the odd numbers or the even numbers:

```
(take 10 (filter even? whole-numbers))
-> (2 4 6 8 10 12 14 16 18 20)

(take 10 (filter odd? whole-numbers))
-> (1 3 5 7 9 11 13 15 17 19)
```

You can follow a sequence while a predicate remains true with

```
(take-while pred coll)
```

For example, to take all the characters in a String up to the first vowel:

```
(take-while (complement #{\a\e\i\o\u}) "the-quick-brown-fox")
-> (\t \h)
```

There are a couple of interesting things happening here:

- Collections also act as functions. So you can read #{\a\e\i\o\u} as either "the set of vowels" or as "the function f(x) which tests to see if x is vowel."
- complement reverses the behavior of another function. The complemented function above tests to see if its argument is *not* a vowel.

The opposite of take-while is

```
(drop-while pred coll)
```

drop-while drops elements from the beginning of a sequence while a predicate is true, and then returns the rest. You could drop-while to drop all leading non-vowels from a string:

```
(drop-while (complement #{\a\e\i\o\u}) "the-quick-brown-fox")
-> (\e \- \q \u \i \c \k \- \b \r \o \w \n \- \f \o \x)
```

If you need both take and drop semantics, split-at and split-with will give you a vector of the results from take and the drop.

```
(split-at index coll)

(split-with pred coll)
```

split-at takes an index, and split-with takes a predicate:

```
(split-at 5 (range 10))
->[(0 1 2 3 4) (5 6 7 8 9)]

(split-with #(<= % 10) (range 0 20 2))
->[(0 2 4 6 8 10) (12 14 16 18)]
```

Sequence Predicates

Filter functions take a predicate and return a sequence. Closely related are the sequence predicates. A sequence predicate asks how some other predicate applies to every item in a sequence. For example, the

```
(every? pred coll)
```

predicate asks if some other predicate is true for every element of a sequence.

```
(every? odd? [1 3 5])
-> true

(every? odd? [1 3 5 8])
-> false
```

A lower bar is set by

```
(some pred coll)
```

some returns the first true value for its predicate, or nil if no element matched.

```
(some even? [1 2 3])
-> true

(some even? [1 3 5])
-> nil
```

Notice that some does not end with a question mark, since it is not quite a predicate. some returns the actual value of the first match instead of true.

The behavior of the other predicates is obvious from their names:

```
(not-every? pred coll)

(not-any? pred coll)
```

Not every whole number is even:

```
(not-every? even? whole-numbers)
-> true
```

But it would be a lie to claim that not any whole number is even.

```
(not-any? even? whole-numbers)
-> false
```

Note that I picked questions to which I already knew the answer. In general, you have to be careful when applying predicates to infinite collections. They might run forever.

Prepared exclusively for Michael Kevin Hagel

Transforming Sequences

Transformation functions transform the values in the sequence. The simplest transformation is map.

```
(map f seq)
```

map takes a source sequence and a function, and returns a new sequence by invoking the function on each element in the source sequence. You can use map to wrap every element in a sequence with an html tag:

```
(map #(format "<p>%s</p>" %) ["the" "quick" "brown" "fox"])
-> ("<p>the</p>" "<p>quick</p>" "<p>brown</p>" "<p>fox</p>")
```

Another common transformation is reduce.

```
(reduce f seq)
```

f is a function of two arguments. reduce applies f on the first two arguments in coll, then applies f to the result plus the third argument, and so on. reduce is useful for functions that "total up" a sequence in some way. You can use reduce to add items:

```
(reduce + (range 1 11))
-> 55
```

Or to multiply them:

```
(reduce * (range 1 11))
3628800
```

The grandaddy of all filters and transformations is the *list comprehension*. A list comprehension creates a list based on an existing list, using set notation–in other words, by stating the properties that the result list must satisfy. In general, a list comprehension will consist of:

- input list(s)
- placeholder variables[1] for elements in the input lists
- predicates on the elements
- an output function that produces output from the elements of the input lists that satisfy the predicate

1. "Variables" in the mathematical sense, not the imperative programming sense. You won't be able to vary them. I humbly apologize for this overloading of the English language.

Prepared exclusively for Michael Kevin Hagel

Of course, Clojure generalizes the notion of list comprehension to *sequence* comprehension. Clojure comprehensions use the for special form[2]:

```
(for [binding-form coll-expr filter-expr? ...] expr)
```

for takes a vector of binding-form/coll-exprs, plus an optional filter-expr, and then yields a sequence of exprs.

List comprehension is more general than functions such as map and filter, and can in fact emulate most of the filtering and transformation functions described above. You can rewrite the map example above as a list comprehension:

```
(for [word ["the" "quick" "brown" "fox"]]
  (format "<p>%s></p>" word))
-> ("<p>the></p>" "<p>quick></p>" "<p>brown></p>" "<p>fox></p>")
```

This reads almost like English: “For [each] word in [a sequence of words] format [according to format instructions].”

Comprehensions can emulate filter using a :when clause. You can pass even? to :when to filter the even numbers:

```
(take 10 (for [n whole-numbers :when (even? n)] n))
-> (2 4 6 8 10 12 14 16 18 20)
```

The real power of for comes when you work with more than one binding expression. For example, you can express all possible positions on a chessboard in algebraic notation by binding both rank and file:

```
(for [file "ABCDEFGH" rank (range 1 9)] (format "%c%d" file rank))
-> ("A1" "A2" ... elided ... "H7 ""H8")
```

Because rank is listed to the right of file in the binding form, rank iterates faster. If you want files to iterate faster, you can reverse the binding order:

```
(for [rank (range 1 9) file "ABCDEFGH"] (format "%c%d" file rank))
("A1" "B1" ... elided ... "G8" "H8")
```

In many languages transformations, filters, and comprehensions do their work immediately. Do not assume this in Clojure. Most functions defer doing anything until you actually try to use the elements.

2. The list comprehension for has nothing to do with the for loop found in imperative languages.

Prepared exclusively for Michael Kevin Hagel

4.3 Lazy and Infinite Sequences

Most Clojure sequences are *lazy*: elements are not calculated until they are needed. Lazy collections can be infinite, and at a very low performance price. *Creating* an infinite collection in Clojure is cheap. You pay the price later, only to the degree you need to, when iterating the collection.

Clojure provides several utilities for dealing with lazy sequences. The lazy-cons and lazy-cat macros combine elements to make a sequence.

```
(lazy-cons first-expr rest-expr)
```

```
(lazy-cat coll & colls)
```

The actual combination is lazy: No operation is performed before it is needed. At the REPL lazy-cons and lazy-cat look almost like their eager brethren cons and concat:

```
(lazy-cons 0 [1 2 3])
-> (0 1 2 3)

(lazy-cat [-3 -2 -1] [0] [1 2 3])
-> (-3 -2 -1 0 1 2 3)
```

The difference is that the lazy versions do not evaluate any of their arguments until forced to do so. You can see this by using both lazy and eager forms with a booby-trapped sequence.

```
(def trapped (concat [1 2] (throw (Exception. "Booby Trap!"))))
-> java.lang.Exception: Booby Trap!
```

The eager form concat evaluates its arguments immediately. But the lazy form lazy-cat postpones evaluating the argument until it needs it:

```
(def trapped (lazy-cat [1 2] (throw (Exception. "Booby Trap!"))))
-> #'user/trapped
```

You can use the booby-trapped collection as long as you don't get close to the booby-trap:

```
(take 1 trapped)
(1)
```

So when should you prefer lazy sequences? Most of the time. Most Clojure sequences are lazy, so you only pay for what you use. More importantly, Clojure makes lazy sequences just as easy to use as eager ones, so choosing a lazy sequence is not a premature optimization. Unless you know that a sequence will immediately be consumed in its entirety, prefer lazyness.

Prepared exclusively for Michael Kevin Hagel

Forcing Sequences

When you are viewing a large sequence from the REPL, you may want to use take to prevent the REPL from evaluating the entire sequence. In other contexts, you may have the opposite problem. You have created a lazy sequence, and you want to force the sequence to evaluate fully.

The problem usually arises when the code generating the sequence has side effects. Consider the following sequence which embeds side effects via println:

```
(def x (for [i (range 1 10)] (do (println i) i)))
#'user/x
```

Newcomers to Clojure are surprised that the code above prints nothing. Since the definition of x does not actually use the elements, Clojure does not evaluate the comprehension to get them. You can force evaluation with

```
(doall coll)
```

doall forces Clojure to walk the elements of a sequence and returns the elements as a result.

```
(doall x)
1
2
-> (1 2)
```

You can also use

```
(dorun coll)
```

dorun walks the elements of a sequence without keeping past elements in memory. As a result, dorun can walk collections too large to fit in memory.

```
user=> (dorun x)
1
2
-> nil
```

The nil return value is a telltale reminder that dorun does not hold a reference to the entire sequence.

The dorun and doall functions help you deal with side effects, while most of the rest of Clojure discourages side effects. You should use these functions rarely. (The Clojure core calls each of these functions only once in about 4000 lines of code.)

Prepared exclusively for Michael Kevin Hagel

4.4 Clojure Makes Java Seq-able

The Seq abstraction of first/next applies to any thing that there can be more than one of. In the Java world, that includes:

- the collections API
- regular expressions
- file system traversal
- XML processing
- relational database results

Clojure wraps these Java APIs, making the sequence library available for almost everything you do.

Seq-ing Java Collections

If you try to apply the sequence functions to Java collections, you will find that they behave as sequences. Collections that can act as sequences are called seq-able. Arrays are seqable:

```
; String.getBytes returns a byte array
(first (.getBytes "hello"))
-> 104

(rest (.getBytes "hello"))
-> (101 108 108 111)
72

(cons 72 (.getBytes "ello"))
-> (72 101 108 108 111)
```

Hashtables and HashMaps are seqable:

```
; System.getProperties returns a Hashtable
(first (System/getProperties))
-> java.runtime.name=Java(TM) SE Runtime Environment

(rest (System/getProperties))
-> (sun.boot.library.path=/System/Library/... etc. ...

; you probably don't want to cons, see below
```

Remember that the sequence wrappers are immutable, even if the underlying Java collection is not. So you cannot update the system properties by consing a new item onto (System/getProperties). All you will do is return a new collection; the existing properties are unchanged.

Since strings are sequences of characters, they also are seqable:

```
(first "Hello")
\H

user=> (rest "Hello")
(\e \l \l \o)

user=> (cons \H "ello")
(\H \e \l \l \o)
```

Clojure will automatically wrap collections in sequences, but it will not automatically rewrap them back to their original type. With most collection types this behavior is intuitive, but with Strings you will often want to convert the result back to a String. Consider reversing a String. Clojure provides reverse:

```
; probably not what you want
(reverse "hello")
-> (\o \l \l \e \h)
```

To convert a sequence back to a string, use (apply str seq):

```
(apply str (reverse "hello"))
-> "olleh"
```

The Java collections are seq-able, but for most scenarios they do not offer advantages over Clojure's built in collections. Prefer the Java collections only in interop scenarios where you are working with legacy Java APIs.

Seq-ing Regular Expressions

Clojure's regular expressions use the java.util.regex library under the hood. At the lowest level, this exposes the mutable nature of Java's Matcher. You can use

```
(re-matcher regexp string)
```

to create a Matcher for a regular expression and a string, and then loop on re-find to iterate over the matches:

Download data_structures/data_structures.clj

```
; don't do this!
(let [m (re-matcher #"\w+" "the quick brown fox")]
  (loop [match (re-find m)]
    (if (nil? match)
      nil
      (do
        (println match)
        (recur (re-find m))))))
```

```
-> the
quick
brown
fox
```

Much better is to use the higher-level re-seq.

```
(re-seq regexp string)
```

re-seq exposes an immutable seq over the matches. This gives you the power of all of Clojure's sequence functions. Try these expressions at the REPL:

```
(re-seq \w+ "the quick brown fox")
-> ("the" "quick" "brown" "fox")

(sort (re-seq \w+ "the quick brown fox"))
-> ("brown" "fox" "quick" "the")

(drop 2 (re-seq \w+ "the quick brown fox"))
-> ("brown" "fox")

(map #(.toUpperCase %) (re-seq \w+ "the quick brown fox"))
-> ("THE" "QUICK" "BROWN" "FOX")
```

re-seq is a great example of how good abstractions reduce code bloat. Regular expressions matches are not a special kind of thing, requiring special methods to deal with them. They are sequences, just like everything else. Thanks to the large number of sequence functions, you get more functionality *for free* than you would likely end up with after a misguided foray into writing regexp-specific functions.

Seq-ing the File System

You can seq over the file system. For starters, you can call java.io.File directly:

```
(.listFiles (File. "."))
-> [Ljava.io.File;@1f70f15e
```

The [Ljava.io.File... is Java's toString() representation for an array of Files. Sequence functions would call seq on this automatically, but the REPL doesn't. So seq it yourself:

```
(seq (.listFiles (File. ".")) )
-> (./data_structures ./exploring ./interop ...)
```

Often, you want to recursively traverse the entire directory tree. Clojure provides a depth-first walk via file-seq. If you file-seq from the sample code directory for this book, you will see a lot of files:

```
(count (file-seq (File. ".")))
-> 104 ; the final number will be larger!
```

What if you want to see only the files that have been changed recently? Write a predicate recently-modified? that checks to see if File was touched in the last half hour:

Download data_structures/data_structures.clj

```
(defn minutes [mins] (* mins 1000 60))

(defn recently-modified? [file]
  (> (.lastModified file) (- (System/currentTimeMillis) (minutes 30))))
```

Give it a try:

```
(filter recently-modified? (file-seq (File. ".")))
-> (./data_structures ./data_structures/data_structures.clj)
```

Since I am working on the data structures examples as I write this, only they show as changed recently. Your results will vary from those above.

Seq-ing a Stream

You can seq over the lines of any Java Reader using line-seq. To get a Reader, you can use Clojure Contrib's duck-streams library. Duck-streams provides a reader function that returns a reader on a stream, file, URL, or URI:

```
; leaves reader open...
(take 2 (line-seq (reader "utils/utils.clj")))
-> ("(ns utils)" "")
```

Since readers can represent non-memory resources that need to be closed, you should wrap reader creation in a with-open. Create an expression that uses the sequence function count to count the number of lines in a file, and uses with-open to correctly close the reader:

```
(with-open rdr (reader "utils/utils.clj")
  (count (line-seq rdr)))
-> 25
```

To make the example a little more useful, add a filter to count only non-blank lines:

```
(with-open rdr (reader "utils/utils.clj")
(count (filter #(re-find #"\\S" %) (line-seq rdr))))
-> 22
```

Using seqs on both the file system and on the contents of individual files, you can quickly create interesting utilities. Create a program that defines these three predicates:

- non-blank? detects non-blank lines
- non-svn? detects files that are not subversion metadata
- clojure-source? detects Clojure source code files

Then, create a clojure-loc function then counts the lines of Clojure code in a directory tree, using a combination of sequence functions along the way: reduce, for, count, and filter.

Download data_structures/data_structures.clj

```
(use 'clojure.contrib.duck-streams)
(defn non-blank? [line] (and (re-find #"\\S" line) true))

(defn non-svn? [file] (not (.startsWith (.toString file) ".svn")))

(defn clojure-source? [file] (.endsWith (.toString file) ".clj"))

(defn clojure-loc [base-file]
  (reduce
   +
   (for [file (file-seq base-file)
         :when (and (clojure-source? file) (non-svn? file))]
     (with-open rdr (reader file)
       (count (filter non-blank? (line-seq rdr)))))))
```

Now, you can use clojure-loc to find out much Clojure code is in Clojure itself:

```
(clojure-loc (java.io.File. "/Users/stuart/repos/clojure"))
-> 4519
```

Those few thousand lines pack quite a punch, because parts such as the sequence library can be recombined in so many ways. Your code can be this powerful, too. The clojure-loc function is very task-specific, but because it is built out of sequence functions and simple predicates, you can easily tweak it to very different tasks.

Seq-ing XML

Clojure can seq over XML data. The examples that follow use this XML file:

Download data_structures/compositions.xml

```
<compositions>
  <composition composer="J. S. Bach">
    <name>The Art of the Fugue</name>
  </composition>
  <composition composer="J. S. Bach">
    <name>Musical Offering</name>
```

```
  </composition>
  <composition composer="W. A. Mozart">
    <name>Requiem</name>
  </composition>
</compositions>
```

The function clojure.xml.parse parses an XML (File|InputStream|URI), and returns the tree of data as a Clojure map, with nested vectors for descendants:

```
(use 'clojure.xml)
(parse (File. "data_structures/compositions.xml"))
-> {:tag :compositions,
    :attrs nil,
    :content [{:tag :composition, ... etc. ...
```

You can manipulate this map directly, or you can use the xml-seq function to view the tree as a seq.

```
(xml-seq root)
```

The example below uses a list comprehension over an xml-seq to extract just the composers:

Download data_structures/data_structures.clj

```
(for [x (xml-seq (parse (File. "data_structures/compositions.xml")))
      :when (= :composition (:tag x))]
  (:composer (:attrs x)))

-> ("J. S. Bach" "J. S. Bach" "W. A. Mozart")
```

The code above demonstrates the generality of seqs, but it only scratches the surface of Clojure's XML support. In Chapter 9, *Clojure-Contrib: Clojure's Standard Library*, on page 125 you will see Clojure's lazy XML and zip-xml support.

4.5 Calling Structure-Specific Functions

Clojure's sequence functions allow you to write very general code. Sometime you will want to be more specific, and take advantage of the characteristics of a specific data structure. Clojure includes functions that specifically target lists, vectors, maps, structs, and sets.

Functions on Lists

Clojure supports the traditional names peek and pop for retrieving the first element of a list and the remainder, respectively:

```
(peek coll)
```

```
(pop coll)
```

Give a simple list a peek and pop:

```
(peek '(1 2 3))
-> 1

(pop '(1 2 3))
-> (2 3)
```

peek is the same as first, but pop is *not* the same as rest. pop will throw an exception if the sequence is empty:

```
(rest ())
-> nil

(pop ())
-> java.lang.IllegalStateException: Can't pop empty list
```

Functions on Maps

Clojure provides several functions for reading the keys and values in a map. keys returns a sequence of the keys, and vals returns a sequence of the values:

```
(keys map)

(values map)
```

Try taking keys and values from a simple map:

```
(keys {:sundance "spaniel", :darwin "beagle"})
-> (:sundance :darwin)

(vals {:sundance "spaniel", :darwin "beagle"})
-> ("spaniel" "beagle")
```

get returns the value for a key, or nil.

```
(get map key)
```

Use your REPL to test that get behaves as expected for keys both present and missing:

```
(get {:sundance "spaniel", :darwin "beagle"} :darwin)
-> "beagle"

(get {:sundance "spaniel", :darwin "beagle"} :snoopy)
-> nil
```

There is an approach even simpler than get. Maps are functions of their keys. So, you can leave out the get entirely, putting the map in function position at the beginning of a form:

Prepared exclusively for Michael Kevin Hagel

```
({:sundance "spaniel", :darwin "beagle"} :darwin)
-> "beagle"

({:sundance "spaniel", :darwin "beagle"} :snoopy)
-> nil
```

Keywords are also functions. They take a collection as an argument, and look themselves up in the collection. Since :darwin and :sundance are keywords, the forms above can be written with their elements in reverse order:

```
(:darwin {:sundance "spaniel", :darwin "beagle"} )
-> "beagle"

(:snoopy {:sundance "spaniel", :darwin "beagle"} )
-> nil
```

If you look up a key in a map and get nil back, you cannot tell if the key was missing from the map, or present with a value of nil. The contains? function solves this problem, by testing for the mere presence of a key.

```
(contains? map key)
```

Create a map where nil is a legal value:

```
(def score {:stu nil :joey 100})
```

:stu is present, but if you see the nil value you might not think so.

```
(:stu score)
-> nil
```

If you use contains?, you can verify that :stu is in the game, although presumably not doing very well:

```
(contains? score :stu)
-> true
```

If nil is a legal value in map, use contains? instead of get to test the presence of a key.

Clojure also provides several functions for building new maps:

- assoc returns map with a key/value pair added
- dissoc returns a map with a key removed
- select-keys returns a map keeping only the keys passed in
- merge combines maps. If multiple maps contains a key, the right-most map wins.

To test these functions, create some song data:

Download data_structures/data_structures.clj

```
(def song {:name "Agnus Dei"
           :artist "Krzystof Pendereci"
           :album "Polish Requiem"
           :genre "Classical"})
```

Next, create various modified versions of the song collection:

```
(assoc song :kind "MPEG Audio File")
-> {:name "Agnus Dei", :album "Polish Requiem",
    :kind "MPEG Audio File", :genre "Classical",
    :artist "Krzystof Penderecki"}

(dissoc song :genre)
-> {:name "Agnus Dei", :album "Polish Requiem",
    :artist "Krzystof Penderecki"}

(select-keys song [:name :artist])
-> {:name "Agnus Dei", :artist "Krzystof Penderecki"}

(merge song {:size 8118166, :time 507245})
-> {:name "Agnus Dei", :album "Polish Requiem",
    :genre "Classical", :size 8118166,
    :artist "Krzystof Penderecki", :time 507245}
```

Remember that song itself never changes. Each of the functions above returns a new collection.

The most interesting map construction function is merge-with.

```
(merge-with merge-fn & maps)
```

merge-with is like merge, except that when two or more maps have the same key, you can specify your own function for combining the values under the key. You can use merge-with and concat to build a sequence of values under each key:

```
(merge-with
 concat
  {:rubble ["Barney"], :flintstone ["Fred"]}
  {:rubble ["Betty"], :flintstone ["Wilma"]}
  {:rubble ["Bam-Bam"], :flintstone ["Pebbles"]})
-> {:rubble ("Barney" "Betty" "Bam-Bam"),
    :flintstone ("Fred" "Wilma" "Pebbles")}
```

Starting with three distinct collections of family members keyed by last name, the code above combines them into a single collection keyed by last name.

Functions on Sets

In addition to the set functions in the clojure namespace, Clojure provides a group of functions in the clojure.set namespace. To use these functions with unqualified names, call (use 'clojure.set) from the REPL. For the examples below, you will also need the following vars:

Download data_structures/data_structures.clj

```
(def languages #{"java" "c" "d" "clojure"})
(def letters #{"a" "b" "c" "d" "e"})
(def beverages #{"java" "chai" "pop"})
(def inventors {"java" "gosling", "clojure" "hickey", "ruby", "matz"})
```

The first group of clojure.set functions perform operations from set theory:

- union returns the set of all elements present in either input set
- intersection returns the set of all elements present in *both* input sets
- difference returns the set of all elements present in the first input set, minus those in the second
- select returns the set of all elements matching a predicate

Write an expression that finds the union of all languages and beverages:

```
(union languages beverages)
-> #{"java" "c" "d" "clojure" "chai" "pop"}
```

Next, try the languages that are not also beverages:

```
(difference languages beverages)
-> #{"c" "d" "clojure"}
```

If you enjoy terrible puns, you will like the fact that some things are both languages *and* beverages:

```
(intersection languages beverages)
-> #{"java"}
```

A surprising number of languages cannot afford a name larger than a single character:

```
(select #(= 1 (.length %))) languages)
-> #{"c" "d"}
```

Set union and difference are part of set theory, but they are also part of *relational algebra*, which is the basis for query languages such as SQL. The relational algebra consists of six primitive operators: set union and

Prepared exclusively for Michael Kevin Hagel

Relational Algebra	Database	Clojure Type System
relation	table	anything setlike
tuple	row	anything maplike

Figure 4.1: Correspondences Between Relational Algebra, Databases, and the Clojure Type System

set difference (described above), plus rename, selection, projection, and cross product.

You can understand the relational primitives by following the analogy with relational databases. (See Figure 4.1). The examples below work against a database of musical compositions. Load the database below before continuing.

Download data_structures/data_structures.clj

```
(def compositions
 #{{:name "The Art of the Fugue" :composer "J. S. Bach"}
   {:name "Musical Offering" :composer "J. S. Bach"}
   {:name "Requiem" :composer "Guiseppe Verdi"}
   {:name "Requiem" :composer "W. A. Mozart"}})
(def countries
 #{{:composer "J. S. Bach" :country "Germany"}
   {:composer "W. A. Mozart" :country "Austria"}
   {:composer "Guiseppe Verdi" :country "Italy"}})
```

The rename function renames keys ("database columns"), based on a map from original names to new names.

```
(rename relation rename-map)
```

Rename the compositions to use a title key instead of name:

```
(rename compositions {:name :title})
  #{{:title "Requiem", :composer "Guiseppe Verdi"}
    {:title "Musical Offering", :composer "J.S. Bach"}
    {:title "Requiem", :composer "W. A. Mozart"}
    {:title "The Art of the Fugue", :composer "J.S. Bach"}}
```

The select function returns maps for which a predicate is true, and is analogous to the WHERE portion of a SQL SELECT.

```
(select pred relation)
```

Write a select expression that finds all the compositions whose title is "Requiem":

```
(select #(= (:name %) "Requiem") compositions)
```

```
#{{:name "Requiem", :composer "W. A. Mozart"}
  {:name "Requiem", :composer "Guiseppe Verdi"}}
```

The project function returns only the portion of the maps that match a set of keys.

```
(project relation keys)
```

project is similar to a SQL SELECT that specifies a subset of columns. Write a projection that returns only the name of the compositions:

```
(project compositions [:name])
#{{:name "Musical Offering"}
  {:name "Requiem"}
  {:name "The Art of the Fugue"}}
```

The final relational primitive, cross product, is the foundation for the various kinds of joins in relational databases. The cross product returns every possible combination of rows in the different tables. You can do this easily enough in Clojure with a list comprehension:

```
(for [m compositions c countries] (concat m c))
-> ... 4 x 3 = 12 rows ...
```

While the cross product is theoretically interesting, you will typically want some subset of the full cross product. For example, you might want to join sets based on shared keys:

```
(join relation-1 relation-2)
```

join the composition names and countries on the shared key :composer

```
(join compositions countries)
-> #{{:name "Requiem", :country "Austria",
      :composer "W. A. Mozart"}
     {:name "Musical Offering", :country "Germany",
      :composer "J. S. Bach"}
     {:name "Requiem", :country "Italy",
      :composer "Guiseppe Verdi"}
     {:name "The Art of the Fugue", :country "Germany",
      :composer "J. S. Bach"}}
```

The analogy between Clojure's relational algebra and a relational database is instructive. Remember, though, that Clojure's relational algebra is a general purpose tool. You can use it on any kind of set-relational data. And while you are using it, you also have the entire power of Clojure and Java at your disposal.

Prepared exclusively for Michael Kevin Hagel

4.6 Wrapping Up

Clojure unifies all kinds of collections under a single abstraction, the sequence. After more than a decade dominated by object-orientated programming, Clojure's sequence library is the "Revenge of the Verbs."

Clojure's sequences are implemented using functional programming techniques: immutable data, recursive definition, and lazy evaluation. In the next chapter, you will see how to use these techniques directly, further expanding the power of Clojure.

Prepared exclusively for Michael Kevin Hagel

Chapter 5

Functional Programming

This is a beta book; this chapter is not yet complete.

Prepared exclusively for Michael Kevin Hagel

Chapter 6

Concurrency

Concurrency is a fact of life, and increasingly a fact of software. There are several important reasons that programs need to do more than one thing at a time:

- Expensive computations need to be execute in parallel on multiple cores (or multiple boxes) in order to complete in a timely manner.
- Tasks that are blocked waiting for a resource need to stand down and let other tasks use available processors.
- User interfaces need to remain responsive while performing long running tasks.
- Operations that are logically independent are easier to implement if the platform can recognize and take advantage of their independence.

Clojure provides three powerful techniques for dealing with the challenges posed by concurrency.

- Software transactional memory (STM) manages coordinated, synchronous changes to shared state.
- Agents provide a more decoupled model, where independent tasks proceed asynchronously.
- Vars managed thread-local bindings, which are useful for a variety of tasks not normally associated with concurrency.

Each of these techniques is discussed in this chapter. Let's get started by saying goodbye to locks, and hello to transactions.

Prepared exclusively for Michael Kevin Hagel

6.1 Coordinating Shared State with STM

Most objects in Clojure are immutable. When you really want mutable data, you must be explicit about it, by creating a mutable *reference* to an immutable object. You create a ref with

```
(ref initial-state)
```

For example, you could create a reference to the current song in your music playlist:

```
(def current-track (ref "Mars, the Bringer of War"))
-> #=(var user/current-track)
```

To get the contents of the reference, you can call

```
(deref ref)
```

The deref function can be shortened to the @ reader macro:

```
(deref current-track)
-> "Mars, the Bringer of War"

@current-track
-> "Mars, the Bringer of War"
```

Notice how the Clojure model fits the real world. A track is an immutable entity. It doesn't change into another track when you are finished listening to it. But the *current* track is a reference to an entity, and it does change.

ref-set

You can change where a reference points with ref-set.

```
(ref-set reference new-value)
```

Call ref-set to listen to a different track:

```
(ref-set current-track "Venus, the Bringer of Peace")
java.lang.IllegalStateException: No transaction running
```

Oops. Because refs are mutable, you must protect their updates. In many languages, you would use a *lock* for this purpose. In Clojure, you can use a *transaction*. Transactions are wrapped in a dosync:

```
(dosync & exprs)
```

Wrap the previous ref-set with a dosync and all is well:

```
(dosync (ref-set current-track "Venus, the Bringer of Peace"))
"Venus, the Bringer of Peace"
```

Prepared exclusively for Michael Kevin Hagel

The current-track reference now refers to a different track.

Like database transactions, Clojure transactions guarantee some important properties:

- Updates are *Atomic*. If you update more than one ref in a transaction, all the updates will happen as a single unit.
- Updates are *Consistent*. Refs can specify validation functions. If any of these functions fail, the entire transaction will fail.
- Updates are *Isolated*. Running transactions cannot see partially completed results from other transactions.

Databases provide the additional guarantee that updates are *Durable*. Because Clojure's transactions are in-memory transactions, Clojure does not guarantee that updates are durable. (If you want a durable transaction in Clojure, you should use a database, or a JMS provider, or some other enterprisey J-acronym.)

Together, the four transactional properties are called the *ACID*. Databases provide ACID, Clojure's software transactional memory provides ACI.

commute

The current-track example is deceptively easy, because updates to the ref are totally independent of any transactional state. Let's build a more complex example, where transactions need to update existing information. A simple chat application fits the bill. First, create a message struct that has a sender and some text:

Download concurrency/chat/chat.clj

```
(defstruct message :sender :text)
```

Now, you can create messages by calling struct:

```
(struct message "stu" "test message")
-> {:sender "stu", :text "test message"}
```

Users of the chat application want to see the most recent message first, so a list is a good data structure. Create a messages reference that points to an initially empty list:

Download concurrency/chat/chat.clj

```
(def messages (ref ()))
```

Now you need a function to add a new message to the front of messages. You could simply deref to get the list of messages, cons the new message, and then ref-set the updated list back into messages:

Download concurrency/chat/chat.clj

```
; bad idea
(defn naive-add-message [msg]
  (dosync (ref-set messages (cons msg @messages))))
```

The naive approach works, but it is inefficient. Consider the following scenario:

- User One attempts to add a message, and gets as far as dereferencing the messages ref.
- User Two begins and completes an update transaction, adding a message.
- User One's transaction attempts to complete, sees that the list was already updated by someone else, and is forced to retry.

User One's transaction will still complete, and the thread will not even have to block. But there is a better option. Why not perform the read and update in a single step? Clojure's commute will apply an update function to a reference object within a transaction:

```
(commute ref update-fn & args...)
```

Using commute instead of ref-set makes the code more readable for humans, and less contentious for threads:

Download concurrency/chat/chat.clj

```
(defn add-message [msg]
  (dosync (commute messages conj msg)))
```

Notice that the update function is conj (short for conjoin), not cons. The important distinction here is argument order:

```
(cons item sequence)
(conj sequence item)
```

The commute function calls the internal update function with the current referenced value as its first argument, as conj expects. If you plan to write your own update functions, they should follow the same structure as conj:

```
(your-func thing-that-gets-updated & optional-other-args)
```

Try adding a few messages to see that the code works as expected:

```
(add-message (struct message "user 1" "hello"))
-> ({:sender "user 1", :text "hello"})

(add-message (struct message "user 2" "howdy"))
({:sender "user 2", :text "howdy"}
```

```
{:sender "user 1", :text "hello"})
```

Among the STM update functions, commute allows for the most concurrency. Of course, there is a tradeoff. Commutes are so named because they must be *commutative*. That is, updates must be able to occur in any order. This gives the STM system freedom to reorder commutes.

Literally speaking, updating messages in the chat application is *not* commutative. The list of messages most certainly has an order, so if two message adds get reversed, the resulting list will not correctly show the order in which the messages arrived.

Practically speaking, message updates are *commutative enough*. STM-based reordering of messages will likely happen on time scales of microseconds or less. For users of the application, there are already reorderings on much larger time scales due to network and human latency. (Think about times that you have "spoken out of turn" in an online chat room because another speaker's message had not reached you yet.) Since these larger reorderings are unfixable, it is reasonable for a chat application to ignore the smaller reorderings that might bubble up from Clojure's STM.

alter

In many settings, updates are not naturally commutative. Instead, you must enforce ordering. For example, consider a counter that returns an increasing sequence of numbers. You might use such a counter to build unique ids in a system. The counter can be a simple reference to a number:

Download concurrency/concurrency.clj

```
(def counter (ref 0))
```

You cannot use commute to update the counter. commute returns the value of the counter at the time of the commute statement, but reorderings could cause the actual end-of-transaction value to be different. This could lead to more than one caller getting the same counter value. Instead, use

```
(alter ref update-fn & args...)
```

alter is a sibling of commute that enforces order and returns the end-of-transaction value.

Download concurrency/concurrency.clj

```
(defn next-counter [] (dosync (alter counter inc)))
```

Try calling next-counter a few times to verify that the counter works as expected:

```
(next-counter)
-> 1

(next-counter)
-> 2
```

Think very carefully when choosing between commute and alter. If you request the wrong semantics, you will get them, and it is very difficult to detect this sort of problem in a test suite. Plan carefully, and augment your normal testing strategies with careful code review.

Adding Validation to Refs

Database transactions achieve consistency through various integrity checks. You can do something similar with Clojure's transactional memory, by specifying a validation function when you create a ref:

```
(ref initial-state validate-fn)
```

The second optional argument to ref is a validation function that can throw an exception to prevent a transaction from completing.

Continuing the chat example, add a validation function to the messages reference that guarantees that all messages have non-nil values for :sender and :text.

Download concurrency/chat/chat.clj

```
(defn validate-message-list [lst]
  (if (not-every? #(and (:sender %) (:text %)) lst)
    (throw (IllegalStateException. "Not a valid message"))))

(def messages (ref () validate-message-list))
```

This validation acts like a key constraint on a table in a database transaction. If the constraint fails, the entire transaction unhappens. Try adding an ill-formed message such as a simple string.

```
(add-message "not a valid message")
-> java.lang.IllegalStateException: Invalid ref state

@messages
-> ()
```

As you can see, add-message fails with an exception, and messages does not change. But where did your error message go? Java exceptions often wrap other exceptions, and in this example you need to unwrap

the exception with getCause() (twice!) to see the original exception that your validator threw:

```
(.. *e getCause getCause)
-> #<java.lang.IllegalStateException: Not a valid message>
```

Refs are great for coordinated access to shared state, but not all tasks require such coordination. For tasks that are more independent, you can use agents.

6.2 Sending Independent Tasks to Agents

Some applications have tasks that can proceed independently with minimal coordination between tasks. Clojure *agents* support this style of task.

Agents have much in common with refs. Like refs, you create an agent by wrapping some piece of initial state:

```
(agent initial-state)
```

Create a counter agent that wraps an initial count of zero:

```
(def counter (agent 0))
-> #=(var user/counter)
```

One you have an agent, you can send the agent a function to update its state:

```
(send agent update-fn & args)
```

Sending to an agent is very much like commuting a ref. Tell the counter to inc:

```
(send counter inc)
-> #<clojure.lang.Agent@23451c74>
```

Notice that the call to send does not return the new value of the agent, returning instead the agent itself. That is because send*does not know* the new value. send tells the agent to update itself on a thread pool, and returns immediately.

You can check the current value of an agent with deref/@, just as you would a ref. By the time you get around to checking the counter, the inc will almost certainly have completed out on the thread pool, raising the value to one:

```
@counter
-> 1
```

If you want to be sure that the agent has completed actions you sent to it, you can call await or await-for:

```
(await & agents)
```

```
(await-for timeout & agents)
```

These functions will cause the current thread to block until all actions sent from the current thread or agent have completed. Be careful with await, as it has no timeout and is willing to wait forever.

Validating Agents and Handling Errors

Agents have other points in common with refs. They also can take a validation function as an optional second argument. Recreate the counter with a validator that ensures it is a number:

```
(use 'clojure.contrib.except) ; for throw-if
-> nil

(def counter (agent 0 #(throw-if (not (number? %)) "not a number")))
-> #=(var user/counter)
```

The clojure.contrib.except library defines

```
(throw-if pred ex-class? format-str & format-args)
```

throw-if is perfect for writing validators. If the new value is not a number, the call to throw-if above throws an exception with the error message not a number.

Try to set the agent to a value that is not a number, by passing an update function that ignores the current value and simply returns a string:

```
(send counter (fn [_] "boo"))
-> #<clojure.lang.Agent@4de8ce62>
```

Everything looks fine (so far) because send still returns immediately. After the agent tries to update itself on a pooled thread, it will enter an exceptional state. You will discover the error when you try to dereference the agent:

```
@counter
-> java.lang.Exception: Agent has errors
```

To discover the specific error (or errors), call agent-errors, which will return a sequence of errors thrown during agent actions:

```
(agent-errors counter)
-> (#<java.lang.IllegalStateException: Invalid agent state>)
```

You can make the agent usable again by calling

```
(clear-agent-errors agent)
```

which returns the agent to its pre-error state. Clear the counter's errors, and verify that it state is the same as before the error occurred.

```
(clear-agent-errors counter)
-> nil

@counter
-> 1
```

Now that you know the basics of agents, let's use them to do some substantial work: a Monte Carlo simulation.

Running a Monte Carlo Simulation

The simulation you will call in this section estimates the value of pi, and is described in Appendix D, on page 134. For now, the important thing is that Monte Carlo simulations converge on a solution the longer you let them run, so they are well-suited for asynchronous execution on an agent.

To load the simulation, use the concurrency.pi library from the sample code:

```
(use 'concurrency.pi)
```

To run the simulation, call run-simulation, passing in the number of iterations you wish to run:

```
(run-simulation 10)
-> {:in-circle 9, :total 10}
```

run-simulation simulates dropping grains of sand onto a square circumscribing a circle. The return value tells the total number of grains, and how many grains landed in the circle. The function guess-pi then guesses pi based on the ratio of the areas of the circle and the square:

```
(guess-pi (run-simulation 100))
-> 2.92
```

2.92 is not a very good guess for pi, because 10 is not nearly enough iterations. It would be nice if you could run additional iterations, and merge those results in with results you already have. run-simulation was written with this in mind. You can call it with two arguments:

```
(run-simulation past-results additional-iterations)
```

Prepared exclusively for Michael Kevin Hagel

Passing past-results as the first argument is perfect for an agent update function. Create a pi-agent agent with an initial value of a hundred thousand, to represent the number of iterations to run:

```
(def pi-agent (agent 100000))
```

Now you can ask pi-agent to run-simulation.

```
(send pi-agent run-simulation)
-> #<clojure.lang.Agent@7e965fb9>
```

This first call to run-simulation will convert the agent value from an iteration count of 100000 to the intermediate data map used by run-simulation. (You number will probably not match those below, since the simulation relies on pseudo-random numbers.)

```
@pi-agent
-> {:in-circle 78779, :total 100000}
```

Check pi-agent's estimate for pi.

```
(guess-pi @pi-agent)
-> 3.15116
```

That's getting better. Ask the agent to run some more iterations:

```
(send pi-agent run-simulation 100000)
-> #<clojure.lang.Agent@7e965fb9>
```

This time, the agent already contains intermediate data, and send adds an extra argument of 100000. send passes arguments through to run-simulation, leading to a function call like:

```
(run-simulation current-intermediate-state 100000)
```

Check to see if the estimate for pi is converging on the actual value:

```
(guess-pi @pi-agent)
-> 3.14378
```

You are gradually converging on pi. You can now send more work to the agent whenever you think the processor has cycles to spare, and the estimate will continue to improve.

This all works because run-simulation was designed to be agent-friendly. In general, agent-friendly functions have the following properties.

- The first argument to the function is of the type to be stored in the agent.
- Additional arguments, if any, are used by the function to create the next agent state.

Prepared exclusively for Michael Kevin Hagel

- The return type of the function is the same as the type of the first argument.

Monte Carlo is very amenable to parallelization, because you can combine the results of iterations on multiple threads. The parallel-guess-pi function shown below lets you assign a swarm of agents to work guessing the value of pi:

Download concurrency/pi/pi.clj

```
(defn parallel-guess-pi [agent-count trials]
  (let [trials (quot trials agent-count)
        agents (for [_ (range agent-count)] (agent trials))]
    (doseq a agents (send a run-simulation))
    (apply await agents)
    (guess-pi (apply merge-with + (map deref agents)))))
```

As you add more agents, parallel-guess-pi should complete more quickly, up to the number of processor cores on your system. See Appendix D, on page 134 for more observations on performance.

Including Agents in Transactions

Transactions cannot have side effects, because Clojure may retry a transaction an arbitrary number of times. However, sometimes you *want* a side effect when a transaction succeeds. Agents provide a solution. If you send an action to an agent from within a transaction, that action will occur exactly once, if and only if the transaction succeeds.

As an example of where this would be useful, consider an agent that writes to a file when a transaction succeeds. You could combine such an agent with the chat example from Section 6.1, *commute*, on page 108 to automatically backup chat messages. First, create a backup-agent that stores the filename to write to.

Download concurrency/concurrency.clj

```
(def backup-agent (agent "messages-backup.clj"))
```

Then, create a modified version of add-message. The new function add-message-with-backup should do two additional things:

- Grab the return value of commute, which is the current database of messages, in a let binding
- While still inside a transaction, send an action to the backup agent that writes the message database to filename. For simplicity, have the action function return filename, so that the agent will use the same filename for the next backup.

Download concurrency/concurrency.clj

```
(def backup-agent (agent "messages-backup.clj"))
```

The new function has one other critical difference: it calls send-off instead of send to communicate with the agent. send-off is a variant of send for actions that expect to block, as a file write might do. send-off actions get their own expandable thread pool. Never send a blocking function, or you may unnecessarily prevent other agents from making progress.

Try adding some messages using add-message-with-backup:

```
(add-message-with-backup (struct message "john" "message one"))
-> ({:sender "john", :text "message one"})

(add-message-with-backup (struct message "jane" "message two"))
-> ({:sender "jane", :text "message two"}
    {:sender "john", :text "message one"})
```

You can check both the in memory messages and the backup file messages-backup to verify that they contain the same structure.

You could enhance the backup strategy in this example in various ways. You could provide the option to backup less often than on every update, or backup only changed information since the last backup.

Since Clojure's STM provides the ACI properties of ACID, and writing to file provides the D (Durability), it is tempting to think that STM plus a backup agent equals a database. This is *not* the case. A Clojure transaction only promises to send(-off) an action to the agent, it does not actually *perform* the action under the ACI umbrella. The moral is simple. If your problem calls for a real database, use a real database.

6.3 Managing Per-Thread State with Vars

When you call def or defn, you create a *dynamic var*, often called just a var. In all the examples so far in the book, you pass an initial value to def, which becomes the *root binding* for the var. For example, the following code creates a root binding for foo of 10:

```
(def foo 10)
-> #=(var user/foo)
```

The binding of foo is shared by all threads. You can check the value of foo on your own thread:

```
foo
-> 10
```

You can also verify foo's value from another thread. Create a new thread, passing it a function that prints foo. Don't forget to start the thread:

```
user=> (.start (Thread. (fn [] (println foo))))
nil
user=> 10
```

In the example above, the call to start() returns nil, and then the value of foo is printed from a new thread.

Most vars are content to keep their root bindings forever. However, you can create a *thread-local* binding for a var with the binding macro:

```
(binding [bindings] & body)
```

Bindings are dynamic, and are visible from any methods that you call while inside the binding.

Structurally, a binding looks a lot like a let. Create a thread-local binding for foo and check its value:

```
(binding [foo 42] foo)
-> 42
```

To see the difference between binding and let, create a simple function that prints the value of foo:

```
(defn print-foo [] (println foo))
-> #=(var user/print-foo)
```

Now, try calling print-foo from both a let and a binding:

```
(let [foo "let foo"] (print-foo))
10

(binding [foo "bound foo"] (print-foo))
bound foo
```

As you can see, the let has no effect outside its own form, and so the first print-foo prints the root binding 10. The binding, on the other hand, stays in effect down any chain of calls that begin in the binding form, so the second print-foo prints bound foo.

Acting At a Distance

Vars intended for dynamic binding are sometimes called special variables. It is good style to name them with leading and trailing asterisks. For example, Clojure uses dynamic binding for thread-wide options such as the standard I/O streams *in*, *out*, an *err*. Dynamic bindings enable *action at a distance*. When you change a dynamic binding,

you can change the behavior of distant functions without changing any function arguments.

One kind of action at a distance is temporarily augmenting the behavior of a function. In some languages this would be classified as Aspect-Oriented Programming; in Clojure it is simply a side effect of dynamic binding.

As an example, imagine that you have a function that performs an expensive calculation. To simulate this, write a function named slow-double that sleeps for a second, and then doubles its input.

Download concurrency/concurrency.clj

```
(defn slow-double [n]
  (Thread/sleep 1000)
  (* n 2))
```

Next, write a function named calls-slow-double that calls slow-double for each item in [1 2 1 2 1 2]:

Download concurrency/concurrency.clj

```
(defn slow-double [n]
  (Thread/sleep 1000)
  (* n 2))
```

Time a call to calls-slow-double. With six internal calls to slow-double, it should take a little over six seconds. Note that you will have to run through the result with dorun; otherwise Clojure's map will outsmart you by being lazy and returning immediately.

```
(time (dorun (calls-slow-double)))
-> "Elapsed time: 6001.437 msecs"
```

Reading the code, you can tell that calls-slow-double is slow because is does the same work over and over again. One times one is two, no matter how many times you ask.

Calculations like slow-double this are good candidates for *memoization*. When you memoize a function, it keeps a cache mapping past inputs to past outputs. If subsequent calls hit the cache, they will return almost immediately. Thus you are trading space (the cache) for time (calculating the function again for the same inputs).

Clojure-contrib provides a memoize, which takes a function and returns a memoization of that function:

```
(memoize function)
```

slow-double is a great candidate for memoization, but it isn't memoized yet, and clients like calls-slow-double already use the slow, unmemoized version. With dynamic binding, this is no problem. Simply create a binding to a memoized version of slow-double, and call calls-slow-binding from within the binding.

Download concurrency/concurrency.clj

```
(use 'clojure.contrib.memoize)
(defn demo-memoize []
  (time
   (dorun
    (binding [slow-double (memoize slow-double)]
      (calls-slow-double)))))
```

With the memoized version of slow-double, calls-slow-double runs three times faster, completing in about two seconds instead of six:

```
(demo-memoize)
"Elapsed time: 2003.236 msecs"
```

This example demonstrates the power and the danger of action at a distance. By dynamically rebinding a function such as slow-double, you change the behavior of *other* functions such as calls-slow-double without their knowledge or consent. There is no distance limit–dynamic rebinding allows you to create effects that are *arbitrarily far away from causes*.

Used occasionally, dynamic binding has great power. But it should not become your primary mechanism for extension or reuse. Functions that use dynamic bindings are not pure functions, and can quickly lose the benefits of Clojure's functional style.

Working with Java Callback APIs

Several Java APIs depend on callback event handlers. GUI frameworks such as Swing use event handlers to respond to user input. XML parsers such as SAX depend on the user implementing a callback handler interface.

These callback handlers are written with mutable objects in mind. Also, they tend to be single-threaded. In Clojure, the best way to meet such APIs halfway is to use dynamic bindings. This will involve mutable references that feel almost like variables (ugh), but because they are used in a single-threaded setting they will not present any concurrency problems.

Clojure provides the set! special form for setting a thread-local dynamic binding:

```
(set! var-symbol new-value)
```

set! should be used rarely. In fact, the only place in the entire Clojure core that uses set! is the Clojure implementation of a SAX ContentHandler.

A ContentHandler receives callbacks as a parser encounters various bits of an XML stream. In nontrivial scenarios, the ContentHandler needs to keep track of *where it is* in the XML stream: the current stack of open elements, current character data, and so on.

In Clojure-speak, you can think of a ContentHandler's current position as a mutable pointer to a specific spot in an immutable XML stream. However, it would be overkill to use Clojure's references, since everything will happen on a single thread. So instead, Clojure's ContentHandler uses dynamic variables and set!. Here is the relevant detail.

```
; redacted from Clojure's xml.clj to focus on dynamic variable usage
(startElement
 [uri local-name q-name #^Attributes atts]
 ; details omitted
 (set! *stack* (conj *stack* *current*))
 (set! *current* e)
 (set! *state* :element))
nil)
(endElement
 [uri local-name q-name]
 ; details omitted
 (set! *current* (push-content (peek *stack*) *current*))
 (set! *stack* (pop *stack*))
 (set! *state* :between)
nil)
```

A SAX parser calls startElement when it encounters an XML start tag. The callback handler updates three thread-local variables. The *stack* is a stack of all the elements the current element is nested inside. The *current* is the current element, and the *state* keeps track of what kind of content we are inside. (This is important primarily when inside character data, which is not shown here.)

endElement reverses the work of startElement, by popping the *stack* and placing the top of the *stack* in *current*.

It is worth noting that 95% of the Java code ever written is written in this style: mutable and single-threadedly oblivious to the possibility of concurrency. Clojure permits this style as an explicit special case, and you should use it for interop purposes only.

Prepared exclusively for Michael Kevin Hagel

The ContentHandler's use of set! does not leak ugliness out into the rest of Clojure. Clojure uses the ContentHandler implementation to build an immutable Clojure structure, which then gets all the benefits of the Clojure concurrency model.

6.4 Wrapping Up

Clojure's concurrency model is the most unique part of the language. The combination of software transactional memory, agents, and dynamic binding that you have seen in this chapter gives Clojure powerful abstractions for all sorts of concurrent systems. It also makes Clojure one of the few languages suited to the coming generation of multi-core computer hardware.

But there's still more. Clojure's macro implementation is easy to learn and use correctly for common tasks, and yet powerful enough for the harder macro-related tasks. In the next chapter, you will see how Clojure is bring macros to mainstream programming.

Prepared exclusively for Michael Kevin Hagel

Chapter 7

Macros

This is a beta book; this chapter is not yet complete.

Prepared exclusively for Michael Kevin Hagel

Chapter 8

Multimethods

This is a beta book; this chapter is not yet complete.

Prepared exclusively for Michael Kevin Hagel

Chapter 9

Clojure-Contrib: Clojure's Standard Library

This is a beta book; this chapter is not yet complete.

Prepared exclusively for Michael Kevin Hagel

Chapter 10

Case Study

This is a beta book; this chapter is not yet complete.

Prepared exclusively for Michael Kevin Hagel

Appendix A

Bibliography

[Lad03] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., 2003.

Prepared exclusively for Michael Kevin Hagel

Appendix B

Clojure Forms

page 58 (new classname)

page 58 (. class-**or**-instance member-symbol & args)
(. class-**or**-instance (member-symbol & args))

page 59 (**import** (& **import**-lists))
; import-list => (package-symbol & class-name-symbols)

page 60 (.. class-**or**-instance form & forms)

page 60 (**doto** class-**or**-inst & member-access-forms)

page 61 (make-array class length)
(make-array class dim & more-dims)

page 62 (aset java-array index value)
(aset java-array index-dim1 index-dim2 ... value)

page 62 (aget java-array index)
(aget java-array index-dim1 index-dim2 ...)

page 62 (alength java-array)

page 62 (to-array sequence)

page 62 (into-array type seq)
(into-array seq)

page 64 (bean java-bean)

page 70 (**proxy** class-**and**-interfaces super-cons-args & fns)

page 72 (add-classpath url)

page 72 (gen-**and**-save-class path name & options)

Prepared exclusively for Michael Kevin Hagel

page 75 (**with-open** name init-form & body)

page 75 (**try** expr* **catch**-clause* **finally**-clause?)
catch-clause -> (**catch** classname name expr*)
finally-clause -> (**finally** expr*)

page 78 (first seq)

page 78 (rest seq)

page 78 (cons elem seq)

page 80 (sorted-set & elements)

page 80 (sorted-map & elements)

page 80 (conj sequence element & elements)

page 80 (conj to-sequence from-sequence)

page 83 (range end)
(range start end)
(range start end step)

page 83 (replicate n x)

page 83 (iterate f x)

page 84 (take n sequence)

page 84 **(repeat x)**

page 84 (cycle coll)

page 84 (interleave & colls)

page 84 (interpose separator coll)

page 85 (str-join separator sequence)

page 85 (list & elements)

page 85 (vector & elements)

page 85 (set elements)

page 85 (hash-map key-1 val-1 ...)

page 86 (take-while pred coll)

page 86 (drop-while pred coll)

page 86 (split-at index coll)

page 86 (split-with pred coll)

Prepared exclusively for Michael Kevin Hagel

page 87 (every? pred coll)

page 87 (some pred coll)

page 87 (not-every? pred coll)

page 87 (not-any? pred coll)

page 88 (map f seq)

page 88 (reduce f seq)

page 89 (**for** (**binding**-form coll-expr filter-expr? ...) expr)

page 90 (**lazy-cons** first-expr rest-expr)

page 90 (lazy-cat coll & colls)

page 91 (doall coll)

page 91 (dorun coll)

page 93 (re-matcher regexp string)

page 94 (re-seq regexp string)

page 97 (xml-seq root)

page 97 (peek coll)

page 98 (pop coll)

page 98 (keys map)

page 98 (values map)

page 98 (get map key)

page 99 (contains? map key)

page 100 (merge-with merge-fn & maps)

page 102 (rename relation rename-map)

page 102 (select pred relation)

page 103 (project relation keys)

page 107 (**dosync** & exprs)

page 109 (commute ref update-fn & args...)

page 110 (alter ref update-fn & args...)

page 111 (ref initial-state validate-fn)

Prepared exclusively for Michael Kevin Hagel

page 112 (agent initial-state)

page 112 (send agent update-fn & args)

page 113 (await & agents)

page 113 (await-**for** timeout & agents)

page 113 (**throw-if** pred ex-class? format-str & format-args)

page 114 (agent-errors counter)
-> (#<java.lang.IllegalStateException: Invalid agent state>)

page 114 (clear-agent-errors agent)

page 118 (**binding** (bindings) & body)

page 120 (memoize function)

page 121 (set! var-symbol new-value)

Prepared exclusively for Michael Kevin Hagel

Appendix C

Editor Support

(Editor support for Clojure is evolving rapidly, so some of the information here may be out-of-date by the time you read this. Check the archives of the mailing list[1] for recent announcements.)

Clojure code is concise and expressive. As a result, editor support is not quite as important as for some other languages. However, you will want an editor than can at least indent code correctly, and can match parentheses.

While writing the book, I used Emacs plus Jeffrey Chu's clojure-mode, available at http://github.com/jochu/clojure-mode. Emacs support for Clojure is quite good, but if you are not already an Emacs user, you might prefer to start with an editor you are familiar with from Figure C.1, on the next page.

The table below summarizes the editor support options I was able to find as of October 2008. (Beta readers: during the beta book period I will be taking a closer look at the various editors in order to add some pointers here. If you have any success or horror stories please post them.)

I used Emacs while writing the book. Bill Clementson has written a quick overview that includes setting up Clojure, Emacs support, and debugging with JSwat.[2]>

1. http://groups.google.com/group/clojure
2. http://bc.tech.coop/blog/081023.html

Prepared exclusively for Michael Kevin Hagel

Editor	Project Name	Project URL
Eclipse	clojure-dev	http://code.google.com/p/clojure-dev/
Emacs	clojure-mode	http://github.com/jochu/clojure-mode
NetBeans	enclojure	http://enclojure.org
Vim	VimClojure	http://kotka.de/projects/clojure/vimclojure.html

Figure C.1: Editor Support for Clojure

Prepared exclusively for Michael Kevin Hagel

Appendix D

A Monte Carlo Simulation in Clojure

This is a beta book; this chapter is not yet complete.

Prepared exclusively for Michael Kevin Hagel

Index

Prepared exclusively for Michael Kevin Hagel

First page of blurb

Prepared exclusively for Michael Kevin Hagel

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Programming Clojure's Home Page
http://pragprog.com/titles/shcloj
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/shcloj.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragprog.com/catalog
Customer Service:	orders@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com

Prepared exclusively for Michael Kevin Hagel