

Extracted from:

Programming Clojure

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

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

Copyright © 2008 The Pragmatic Programmers, LLC.

All rights reserved.

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

The
Pragmatic
Programmers

Programming Clojure



Stuart Halloway

Edited by Susannah Davidson Pfalzer



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 © 2009 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.

P1.0 printing, May 2009

Version: 2009-5-30

If you want to also reload any namespaces that `examples.exploring` refers to, you can pass `:reload-all`:

```
(use :reload-all 'examples.exploring)
```

It is idiomatic to import Java classes and use namespaces at the top of a source file, using the `ns` macro:

```
(ns name & references)
```

The `ns` macro sets the current namespace (available as `*ns*`) to `name`, creating the namespace if necessary. The `references` can include `:import`, `:require`, and `:use`, which work like the similarly named functions to 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 `examples/exploring.clj`

```
(ns examples.exploring
  (:use examples.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, 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 `examples/exploring.clj`

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

```
(is-small? 50)
⇒ "yes"
```

If the first 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](#) examples/exploring.clj

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

```
⇒ (is-small? 50000)
    "no"
```

The `when` and `when-not` control flow macros are built on top of `if` and are described in Section 7.2, *when and when-not*, on page 219.

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](#) examples/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"
```

This is an example of a *side effect*. The `println` doesn’t 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.

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 bindings and returns control to the top of the loop. For example, the following loop/recur returns a countdown:

[Download](#) examples/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 to an empty vector and binds x to 5. Since x is not zero, recur then rebinds the names x and result:

- result binds to the previous result conjoined with the previous x.
- x binds to the decrement of the previous x.

Control then returns to the top of the loop. Since x is again not zero, the loop continues, accumulating the result and decrementing x. Eventually, x reaches zero, and the if terminates the recurrence, returning result.

Instead of using a loop, you can use recur back to the top of a function. This makes it simple to write a function whose entire body acts as an implicit loop:

[Download](#) examples/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. But you may not use it very often, because many common recursions are provided by Clojure's sequence library.

For example, countdown could also be expressed as any of these:

```
⇒ (into [] (take 5 (iterate dec 5)))
   [5 4 3 2 1]

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

⇒ (vec (reverse (rest (range 6))))
   [5 4 3 2 1]
```

Do not 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 119. Clojure *will not* perform automatic tail-call optimization (TCO). However, it will optimize calls to `recur`. Chapter 5, *Functional Programming*, on page 149 defines TCO and explores recursion and TCO in detail.

At this point, you have seen quite a few language features but still no variables. Some things really do vary, and Chapter 6, *Concurrency*, on page 179 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.

2.6 Where's My for Loop?

Clojure has no `for` loop and no direct mutable variables.⁸ 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 (sort of) randomly, find a method with some `for` loops and variables, and port it to Clojure. I opened the Apache Commons project, which is 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 for a method that had multiple `for` loops and local variables and found `indexOfAny`:

```
Download snippets/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;
    }
}
```

8. Clojure provides *indirect* mutable references, but these must be explicitly called out in your code. See Chapter 6, *Concurrency*, on page 179 for details.

```

    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;
}

```

`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, as counted by David A. Wheeler's SLOCCount.⁹

Now let's build a Clojure `index-of-any`, step by step. If we just wanted to find the matches, we could use a Clojure filter. But we want to find the *index* of a match. So, we create `indexed`,¹⁰ a function that takes a collection and returns an indexed collection:

[Download](#) `examples/exploring.clj`

```
(defn indexed [coll] (map vector (iterate inc 0) coll))
```

`indexed` returns a sequence of pairs of the form `[idx elt]`. The expression `(iterate inc 0)` produces the indexes, and the `coll` argument provides the elements. Try indexing a string:

```

=> (indexed "abcde")
    ([0 \a] [1 \b] [2 \c] [3 \d] [4 \e])

```

Next, we want to find the indices of all the characters in the string that match the search set.

9. <http://www.dwheeler.com/sloccount/>

10. The `indexed` function already exists as part of `clojure-contrib`, but I am reimplementing it here for fairness of comparison.

Create an index-filter function that is similar to Clojure's `filter` but that returns the indices instead of the matches themselves:

[Download](#) examples/exploring.clj

```
(defn index-filter [pred coll]
  (when pred
    (for [[idx elt] (indexed coll)] :when (pred elt) idx)))
```

Clojure's `for` is *not* a loop but a sequence comprehension (see Section 4.2, *Transforming Sequences*, on page 124). The index/element pairs of `(indexed coll)` are bound to the names `idx` and `elt` but only when `(pred elt)` is true. Finally, the comprehension yields the value of `idx` for each matching pair.

Clojure sets are functions that test membership in the set. So, you can pass a set of characters and a string to `index-filter` and get back the indices of all characters in the string that belong to the set. Try it with a few different strings and character sets:

```
⇒ (index-filter #{\a \b} "abcdbbb")
   (0 1 4 5 6)
```

```
⇒ (index-filter #{\a \b} "xyz")
   nil
```

At this point, we have accomplished *more* than the stated objective. `index-filter` returns the indices of all the matches, and we need only the first index. So, `index-of-any` simply takes the first result from `index-filter`:

[Download](#) examples/exploring.clj

```
(defn index-of-any [pred coll]
  (first (index-filter pred coll)))
```

Test that `index-of-any` works correctly with a few different inputs:

```
⇒ (index-of-any #{\z \a} "zzabyycdxx")
   0
⇒ (index-of-any #{\b \y} "zzabyycdxx")
   3
```

The Clojure version is simpler than the imperative version by every metric (see Figure 2.2, on the following page). What accounts for the difference?

- The imperative `indexOfAny` must deal with several special cases: null or empty strings, a null or empty set of search characters, and the absence of a match. These special cases add branches and exits to the method. With a functional approach, most of these kinds of special cases just work without any explicit code.

Metric	LOC	Branches	Exits/Method	Variables
Imperative Version	14	4	3	3
Functional Version	6	1	1	0

Figure 2.2: Relative complexity of imperative and functional `indexOfAny`

- The imperative `indexOfAny` introduces local variables to traverse collections (both the string and the character set). By using higher-order functions such as `map` and sequence comprehensions such as `for`, the functional `index-of-any` avoids all need for variables.

Unnecessary complexity tends to snowball. For example, the special case branches in the imperative `indexOfAny` use the magic number `-1` to indicate a nonmatch. Should the magic number be a symbolic constant? Whatever you think the right answer is, *the question itself disappears* in the functional version. While shorter and simpler, the functional `index-of-any` is also *vastly more general*:

- `indexOfAny` searches a string, while `index-of-any` can search any sequence.
- `indexOfAny` matches against a set of characters, while `index-of-any` can match against any predicate.
- `indexOfAny` returns the first match, while `index-filter` returns all the matches and can be further composed with other filters.

As an example of how much more general the functional `index-of-any` is, you could use code we just wrote to find the third occurrence of “heads” in a series of coin flips:

```
(nth (index-filter #{:h} [:t :t :h :t :h :t :t :h :h])
  2)
8
```

So, it turns out that writing `index-of-any` in a functional style, without loops or variables, is simpler, less error prone, and more general than the imperative `indexOfAny`.¹¹ On larger units of code, these advantages become even more telling.

11. 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 language. See <http://functionaljava.org/> for more information.

2.7 Metadata

The Wikipedia entry on metadata¹² begins by saying that metadata is “data about data.” 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 flushed to the database is metadata.

You can add metadata to a collection or a symbol using the `with-meta` function:

```
(with-meta object metadata)
```

Create a simple data structure, then use `with-meta` to create another object with the same data but its own metadata:

```
(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
```

The `=` tests for value equality, like Java’s `equals`. To test reference equality, use `identical?`:

```
(identical? obj1 obj2)
```

You can prove that `stu` and `serializable-stu` are different objects by calling `identical?`:

```
(identical? stu serializable-stu)
⇒ false
```

`identical?` is equivalent to `==` in Java.

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}
```

12. <http://en.wikipedia.org/wiki/Metadata>

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}
```

When you create a new object based on an existing object, the existing object's metadata flows to the new object. For example, you could add some more information to `serializable-stu`. The `assoc` function returns a new map with additional key/value pairs added:

```
(assoc map k v & more-kvs)
```

Use `assoc` to create a new collection based on `serializable-stu`, but with a `:state` value added:

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

`stu-with-address` has the new key/value pair, and it also takes on the metadata from `serializable-stu`:

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

In addition to adding metadata to your own data structures, you can also pass metadata to the Clojure compiler using the reader metadata macro.

Reader Metadata

The Clojure language itself uses metadata in several places. For example, vars have a metadata map containing documentation, type information, and source information. Here is the metadata for the `str` var:

```
⇒ (meta #'str)
   {:ns #<Namespace clojure.core>,
    :name str,
    :file "core.clj",
    :line 313,
    :arglists ([] [x] [x & ys]),
    :tag java.lang.String,
    :doc "With no args, ... etc."}
```

Some common metadata keys and their uses are shown in Figure 2.3, on page 79.

Much of the metadata on a var is added automatically by the Clojure compiler. To add your own key/value pairs to a var, use the metadata reader macro:

```
#^metadata form
```

For example, you could create a simple shout function that upcases a string and then document that shout both expects and returns a string, using the `:tag` key:

```
; see also shorter form below
(defn #^{:tag String} shout [#^{:tag String} s] (.toUpperCase s))
⇒ #'user/shout
```

You can inspect shout's metadata to see that Clojure added the `:tag`:

```
^#'shout
{:ns #<Namespace user>,
 :name shout,
 :file "NO_SOURCE_FILE",
 :line 57,
 :arglists ([s]),
 :tag java.lang.String}
```

You provided the `:tag`, and Clojure provided the other keys. The `:file` value `NO_SOURCE_FILE` indicates that the code was entered at the REPL.

You can also pass a nonstring to shout and see that Clojure enforces the `:tag` by attempting to cast the argument to a string:

```
(shout 1)
⇒ java.lang.ClassCastException: \
   java.lang.Integer cannot be cast to java.lang.String
```

Because `:tag` metadata is so common, you can also use the short-form `#^Classname`, which expands to `#^{:tag Classname}`. Using the shorter form, you can rewrite shout as follows:

```
(defn #^String shout [#^String s] (.toUpperCase s))
⇒ #'user/shout
```

If you find the metadata disruptive when you are reading the definition of a function, you can place the metadata last. Use a variant of `defn` that wraps one or more body forms in parentheses, followed by a metadata map:

```
(defn shout
  ([s] (.toUpperCase s))
  {:tag String})
```

Metadata Key	Used For
:arglists	Parameter info used by doc
:doc	Documentation used by doc
:file	Source file
:line	Source line number
:macro	True for macros
:name	Local name
:ns	Namespace
:tag	Expected argument or return type

Figure 2.3: Common metadata keys

It is important to note that the metadata reader macro is *not the same* as `with-meta`. The metadata reader macro adds metadata for the compiler, and `with-meta` adds metadata for your own data:

```
⇒ (def #{:testdata true} foo (with-meta [1 2 3] {:order :ascending}))
    #'user/foo
```

When Clojure reads the previous form, the compiler adds `:testdata` to the metadata for the `var` `foo`:

```
(meta #'foo)
{:ns #<Namespace user>, :name foo, :file "NO_SOURCE_FILE",
 :line 6, :testdata true}
```

The `with-meta` adds `:order` to the *value* `[1 2 3]`, which is then bound to `foo`:

```
⇒ (meta foo)
{:order :ascending}
```

As a general rule, use the metadata reader macro to add metadata to vars and parameters. Use `with-meta` to add metadata to data.¹³

2.8 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 purely functional code, and yet you can easily introduce side effects

13. As with any good rule, there are exceptions. Inside a macro definition you may need to use `with-meta` to add metadata to vars. See Section 7.5, *Making a Lancet DSL*, on page 235 for an example.

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, new titles and other offerings.

Buy the Book

If you liked this eBook, 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

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)