Extracted from:

Getting Clojure

Build Your Functional Skills One Idea at a Time

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Getting Clojure

Build Your Functional Skills One Idea at a Time

Russ Olsen

edited by Michael Swaine

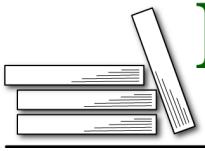
Getting Clojure

Build Your Functional Skills One Idea at a Time

Russ Olsen

The Pragmatic Bookshelf

Raleigh, North Carolina



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

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Development Editor: Michael Swaine
Copy Editor: Candace Cunningham
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

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

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

Printed in the United States of America.
ISBN-13: 978-1-68050-300-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—May 2018

To

Mikey

Felicia

Jackson

Charlie & Jennifer

Tim & Emily & Evan

Nicholas & Jonathan

Meg & Alex & Zachary

and

Scott Downie

The future is in your hands.

This is where the fun starts: In this chapter I'm going to take you through the basic elements of Clojure, at the *just enough knowledge to be dangerous* level. By the time you get to the end of this chapter you'll be familiar with Clojure's very simple syntax and you'll know about many of the basic Clojure data types. Most importantly, you will be able to create and use the fundamental unit of Clojure code, the function. A couple of things to keep in mind as you read:

First, I can't actually teach you Clojure. All I can do is to serve as your guide, to take you around, show you the sights, point out the cool bits, and warn you about the occasional pothole. It's up to you, armed with patience and a keyboard, to do the real work of exploring the land of Clojure.

Second, know that Clojure is worth the effort. Clojure enables you to write clean, compact code that does what you want it to do. In fact, as we go along you'll discover that the Clojure way of programming provides enormous power wrapped up in lovely, elegant code. Let's get started.

The Very Basics

To get started you need to install some development tools. There's a wide selection of Clojure development environments and build tools available—everything from the clj tool that comes packaged with Clojure starting with version 1.9 to the IntelliJ-based Cursive to Emacs and Cider and boot.[1] [2] [3] [4] [5] [6] But in this book we're mostly going to stick to the popular Clojure development tool *Leiningen*. So if you haven't already, head over to the Leiningen website and follow the installation instructions for your operating system.[7] While you're there, you might also note that it's pronounced LINE-ing-en.

By tradition, the first program you write when learning a programming language simply prints a greeting. Here's the Clojure version:

hello/examples.clj

```
(println "Hello, world!")   ; Say hi.
```

1. https://clojure.org/guides/deps_and_cli
2. https://www.jetbrains.com/idea
3. https://cursive-ide.com
4. https://www.gnu.org/software/emacs
5. https://github.com/clojure-emacs/cider
6. https://github.com/boot-clj/boot
7. https://leiningen.org

To keep things simple, we'll take our first stab at Hello, World in the Clojure *REPL*, a handy utility that lets you type in code and see it evaluated right here, right now. The command to start a REPL with Leiningen is as follows:

```
$ lein repl
```

And once you have the REPL running you can type in this code:

```
user=> (println "Hello, world!")   ; Say hi.
```

And see the familiar greeting:

```
Hello, world!
nil
```

Don't fret about the nil; that's just the value returned by println after it does its thing, which the REPL helpfully printed for us.

REPL Who?

The REPL is one of the few programs whose name is its algorithm. All the REPL does is read some code—the code you type in—evaluate the code, print the result, and then loop back to read some more code: *R*ead. *E*valuate. *P*rint. *L*oop.

One of the things that has made Hello, World such a popular first program is just how much we can learn from that single line of code. Looking at our Clojure Hello, World, we can work out that in Clojure strings come "wrapped in double quotes".

We can also see that comments start with a semicolon and run to the end of the line. Typically Clojure programmers will use a single semicolon when they add a comment to the end of a line with some code—as we did in the example—but will double up on the semicolons if the comment is all alone on its own line:

hello/examples.clj

```
;; Do two semicolons add up to a whole colon?

(println "Hello, world!")   ; Say hi
```

More subtly, we can deduce that Clojure treats simple, unadorned names like println as identifying things that get looked up. Thus our little program only worked because println is the name of a predefined function, one that comes to us courtesy of Clojure itself. As you might expect, Clojure predefines a whole range of other handy functions. There is, for example, str, which takes any number of values, converts them to strings, and concatenates the whole thing together:

```
(str "Clo" "jure")              ; Returns "Clojure".
```

```
(str "Hello," " " "world" "!")     ; Returns the string "Hello, world!"
(str 3 " " 2 " " 1 " Blast off!")  ; Fly me to the Moon!
```

There is also count, which will tell you how long your string is:

```
(count "Hello, world") ; Returns 12.
(count "Hello")        ; Returns 5.
(count "")             ; Returns 0.
```

Clojure also comes with a number of predefined constants. For example, we have the Boolean siblings true and false:

```
(println true)     ; Prints true...
(println false)    ; ...and prints false.
```

There is also nil, which is Clojure's version of the "nobody's home" value, known in some languages as null or None:

```
(println "Nobody's home:" nil)   ; Prints Nobody's home: nil
```

Note that println will print just about anything you throw at it, so that if we run this:

```
(println "We can print many things:" true false nil)
```

we'll see

```
We can print many things: true false nil
```

You've probably noticed something odd about the parentheses in a Clojure function call: they are on the outside. It's

```
(println "Hello, world!")
```

not

```
println("Hello, world!")
```

If you're coming to Clojure from a more traditional programming language, those parentheses will look out of place. There is a method to the Clojure syntax madness, which we'll return to in *Read and Eval*. For now let's just note that the Clojure syntax for making something happen—such as calling a function— is to wrap the something in round parentheses, and move on.

Arithmetic

Another thing that usually comes early in learning a programming language is figuring out how to do basic arithmetic. Clojure's approach to doing math

is refreshingly—and perhaps a little disconcertingly—simple. To see what this means, let's add a couple of numbers together:

```
(+ 1900 84)
```

Run the expression in that example through the REPL and you will get back `1984`. You do multiplication, subtraction, and division in the same way:

```
(* 16 124)   ; Gives you 1984.
(- 2000 16)  ; 1984 again.
(/ 25792 13) ; 1984 yet again!
```

As you might expect, you can assemble these basic math operations into arbitrarily complex expressions. Thus you can get the average of `1984` and `2010` with this:

```
(/ (+ 1984 2010) 2)
```

Arithmetic in Clojure can be a bit disorienting at first, a disorientation that can be summed up by the question *Why is the + first?* or perhaps *What happened to my nice infix operators?* The answer is that Clojure is trading some convenience, in the form of the familiar infix operators, for simplicity. By treating the arithmetic operators like ordinary functions, Clojure manages to keep the syntax of the language uniform. In Clojure, no matter what you are doing, you do it by saying

```
(verb argument argument argument...)
```

This means that in the same way we print the string `"hello"` with `(println "hello")`, we add two numbers with `(+ 1982 2)` and we divide them with `(/ 25792 13)`. It's always the thing we want to do, followed by any arguments, all wrapped in round parentheses.

Conveniently, the basic math operators/functions take a variable number of arguments. Thus we can add up a bunch of numbers with this:

```
(+ 1000 500 500 1)  ; Evaluates to 2001.
```

or do a running subtraction with this:

```
(- 2000 10 4 2)     ; Evaluates to 1984;
```

There is one other twist lurking in the math functions, specifically in the `/` (division) function. Many programming languages, when asked to divide one integer by another, will give you back a truncated integer result. For example, in Ruby or Java when you divide 8 by 3 you get 2. Not so in Clojure, where `(/ 8 3)` will give you `8/3`, which is a *ratio*, one of Clojure's built-in data types.

To get the familiar integer truncating behavior, you need to use the quot—short for quotient—function. So one way to get 2 is to write (quot 8 3).

By default, Clojure turns unadorned numeric literals like 8 and 3 and 4976 into integers. If you are interested in numbers with decimal points, Clojure also offers the familiar floating-point notation. Here's our averaging expression again, this time using floating-point numbers:

```
(/ (+ 1984.0 2010.0) 2.0)
```

Clojure also provides a sensible set of numeric promotions, so that if you add an integer to a floating-point number, perhaps (+ 1984 2010.0), you will get back a floating-point number—in this case 3994.0—for your trouble.

Not Variable Assignment, but Close

Once you get beyond the *add a few numbers together* stage you naturally start looking for a way to hang a name on the result. The most straightforward way to do that in Clojure is with def:

```
(def first-name "Russ")
```

There are very few surprises in using def. You give it an identifier—Clojure calls this a *symbol*—and a value, and def will associate, or *bind*, the symbol to the value. In this example the symbol is first-name and the value is the string "Russ". The value that you supply to def gets evaluated, so it can be any expression. So evaluating this

```
(def the-average (/ (+ 20 40.0) 2.0))
```

Will bind 30.0 to the-average.

One thing that you might find surprising is that it's the-average and not theAverage or the_average or even TheAverage. While the Clojure language is gloriously easygoing when it comes to the characters you can use in a symbol—this&that|other and

Much=M*re! are both fine—Clojure programmers have adopted the all-lower-case-with-words-separated-by-dashes convention—also known as *kebab case*—when picking symbols, so it's first-name and the-average.

A note of caution: def is great when you're just playing around or debugging in the REPL, but it's not the direct analog of traditional variable assignment that it seems. We'll get back to the distinction in *Def, Symbols, and Vars*, but for now we'll put that aside and continue to def things with wild abandon.

Symbolic Rules?

As I say, there are very few rules about the characters that can go into a symbol. But there are some: You can't, for example include parentheses, square brackets, or braces in your symbols since these all have a special meaning to Clojure. For the same reason, you can't use the @ and ^ characters in your symbols.

There are also some special rules for the *first* character of your symbols: you can't kick your symbol off with a digit—it would be too easily confused with a number—and symbols that start with a colon are not actually symbols but rather *keywords*, which we'll talk about in *Maps, Keywords, and Sets*.

A Function of Your Own

Let's return to our Hello, World example and see if we can turn our one-liner into something more worthy of the name *program*. We can do this by wrapping it with the fundamental unit of Clojure code, the function:

```
(defn hello-world [] (println "Hello, world!"))
```

Once you have hello-world defined you can call it just like any other Clojure function, so that if you run

```
(hello-world)
```

You will see Hello, world! printed.

As with the original, you can learn a lot from this new version of Hello, World. For example, you can see that the function definition kicks off with defn instead of def and is wrapped in its own set of parentheses *on the outside*. Inside the defn we have the function parameters, set off with square brackets, []. Since our hello-world function doesn't have any parameters, there is nothing inside of its brackets.

While we wrote the original version of Hello, World entirely on one line, we could have spread the defn over a couple of lines:

```
(defn hello-world []
  (println "Hello, world!"))
```

It's all the same to the compiler because Clojure mostly ignores whitespace.

Clojure programmers do, however, have opinions about whitespace. By convention, you can either write a short function on a single line or spread it out over a couple of lines as we did in the last example. Longer functions should take up as many lines as they need. Clojure programmers also have a strong opinion about indentation, one that we followed in the example: each level of indentation is done with two spaces. There are exceptions to the two-space rule, mostly around lining up function arguments and the like. But for the moment we'll stick to two spaces. And note it's always *spaces*, no tabs allowed.

Sans Tabs?

Why no tabs? Because one of the great mysteries of programming is the exchange rate between tabs and spaces. Is it four spaces to a tab? Eight? Three? It's safer to stick to spaces.

Writing a function with a parameter or two is also straightforward: just put the parameter names in the brackets and then use them inside the function body. Here's a greeting function that takes a single parameter:

```
(defn say-welcome [what]
  (println "Welcome to" what))
```

The say-welcome function takes one parameter, called what, and prints an appropriate greeting. Calling your new function is like calling the println function, so that if you do this:

```
(say-welcome "Clojure")
```

you should see a friendly greeting:

```
Welcome to Clojure
```

Happily, we can rely on println to supply the spaces around the values that it prints so that we see "Welcome to Clojure" and not "Welcome toClojure."

Now that we have the basic function-building mechanics down, let's see if we can create a function that does something useful:

```
;; Define the average function.

(defn average [a b]
  (/ (+ a b) 2.0))

;; Call average to find the mean of 5.0 & 10.0.

(average 5.0 10.0)     ; Returns 7.5
```

The average function takes a couple of numbers and returns their arithmetic mean. There are three things to note about the average function. The first is the comma between the two parameters: it's not there. In contrast to many programming languages, Clojure never requires you to sprinkle commas in when you're writing a sequence of items such as the parameter list of a function. A bit of whitespace between the items is plenty. But if you really miss the commas, you can put them in: Clojure treats commas *as whitespace*. Clojure programmers mostly dispense with commas.

The second thing to note about average is that there is no explicit return statement. Clojure functions just return whatever they compute. More precisely, they return the *last* thing computed by the function. We need the qualifier because you can have more than one expression inside of your function body. Here, for example, is a variation on average that has a four-expression body:

```
(defn chatty-average [a b]
  (println "chatty-average function called")
  (println "** first argument:" a)
  (println "** second argument:" b)
  (/ (+ a b) 2.0))
```

You can probably guess what happens when you evaluate chatty-average:

```
(chatty-average 10 20)
```

Each expression inside the body gets evaluated in turn, so that you would see this:

```
chatty-average function called
** first argument: 10
** second argument: 20
```

Since the last expression supplies the return value, the function returns 15.0.

The final thing to note about average is that there are no type declarations, nothing stating explicitly that a and b must now and forever be numbers. Nor is there anything declaring what the function returns. In the great "static versus dynamic typing" trade-off, Clojure has chosen the flexibility and terseness of dynamic typing.