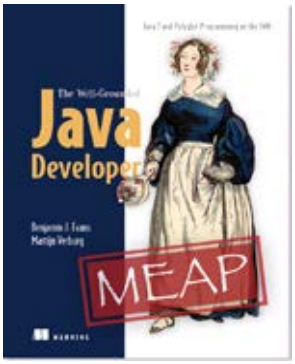
MANNING PUBLICATIONS



The Well-Grounded Java Developer
Benjamin J. Evans and Martijn Verburg

As a language, Clojure is arguably the most different from Java. In this article, based on The Well-Grounded Java Developer, the authors show you Clojure's most important conceptual variations from Java—the treatment of state and variables.

Dipping Your Toes in Clojure

We'll look at one of Clojure's most important conceptual variations from Java. This is the treatment of state and variables. As you can see in figure 1, Java (and Groovy and Scala) has a model of memory and state that involves a variable being a "box" (really, a memory location) with contents that can change over time.

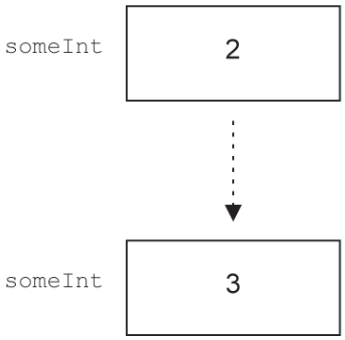


Figure 1 Picture of memory use in an imperative language

Clojure is a little bit different—the important concept is that of a value. Values can be numbers, strings, vectors, maps, sets, or a number of other things. Once created, values never alter. This is really important, so we'll say it again. Once created, Clojure values cannot be altered—they are *immutable*.

This means that the imperative language model of a box that has contents that change is not the way Clojure works. Figure 2 shows how Clojure deals with state and memory – by creating an association between a name and a value.

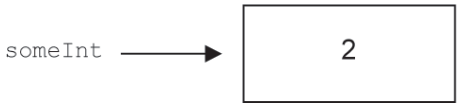


Figure 2 Clojure memory use

This is called binding, and is done using the "special form" `(def)`. Special forms are the Clojure equivalent of a Java keyword, but be aware that Clojure has a different meaning for the term "keyword", which we'll encounter later. The syntax for `(def)` is:

```
(def <name> <value>)
```

Don't worry that the syntax looks a little weird – this is entirely normal for Lisp syntax and you'll get used to it really quickly. Just pretend that the brackets were arranged slightly differently and that we're calling a method like this:

```
def(<name>, <value>)
```

Let's demonstrate `(def)` with a time-honored example, which uses the Clojure interactive environment.

Example—"Hello World" in Clojure

Go into the directory where you installed Clojure and run:

```
java -cp clojure.jar clojure.main
```

This brings up the user prompt for the Clojure Read-Evaluate-Print Loop (the REPL). This is the interactive session, which is where a developer will typically spend quite a lot of time when developing Clojure code.

The `user=>` part is the Clojure prompt for our session, which can be thought of a bit as an advanced debugger or a command line:

```
user=> (def hello (fn [] "Hello world"))
#'user/hello
user=> (hello)
"Hello world"
```

In this code, we're starting off by binding the identifier `hello` to a value. `(def)` always binds identifiers (which Clojure calls "symbols") to values. Behind the scenes, it will also create an object—called a var—which represents the binding (and the name of the symbol).

What is the value we're binding to? It's the value `(fn [] "Hello world")`. This is a function, which is a genuine value (and, therefore, immutable) in Clojure. It's a function that takes no arguments and returns the string "Hello world."

After binding it, we're then executing it via `(hello)`. This causes the Clojure runtime to print out the results of evaluating the function, which is "Hello world."

Getting started with the REPL

The REPL allows the user to enter Clojure code and execute Clojure functions. It is an interactive environment and the results of earlier evaluations are still around. This enables a type of programming called *exploratory programming*. It basically means that the developer can experiment with code. In many cases, the right thing to do is to play around—building up larger and larger functions once the building blocks are correct. At this point, you should enter the "Hello World" example and see that it behaves as described. One of the first things to point out is that binding a symbol to a value can be changed by another call to def:

```
user=> (hello)
"Hello world"
user=> (defn hello [] "Goodnight Moon")
#'user/hello
user=> (hello)
"Goodnight Moon"
```

Notice how the original binding for `hello` was still in play until we changed it—this is a key feature of the REPL. Remember that all we're really doing is evaluating functions. There is still state in terms of which symbols are bound to which values, which persists between lines the user enters.

The ability to change the value a symbol is bound to is Clojure's alternative to mutating state. Rather than changing the contents of a "memory box" over time, Clojure allows a symbol to be bound to different immutable values at different points in time. Another way of saying this is that the var can point at different values during the lifetime of a program. An example can be seen in figure 3.

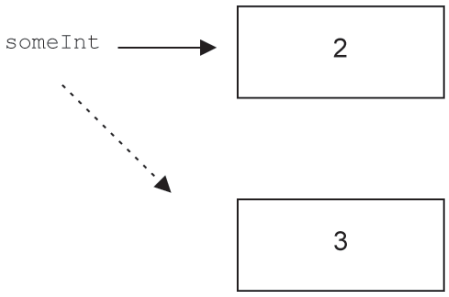


Figure 3 Clojure bindings changing over time

> **NOTE** This distinction between the mutable state and different bindings at different times is subtle but is a very important concept to grasp.

We've also slipped in another Clojure concept in the last code snippet—`(defn)`, the define function macro. Macros are one of the key concepts of Lisp-like languages. The central idea is that there should be as little distinction between built-in constructs and ordinary code as possible.

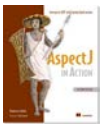
Macros allow a programmer to create forms that behave like built-in syntax. The creation of macros is an advanced topic, but mastering their creation will allow a Clojure developer to produce incredibly powerful tools. This means that the true language primitives of the system (the special forms) can be used to build up the core of the language in such a way that the user doesn't really notice the difference. The `(defn)` macro is an example of this—just a slightly easier way to bind a function value to a symbol (and create a suitable var, of course).

Summary

As a language, Clojure is arguably the most different from Java. Its Lisp heritage, emphasis on immutability, and different approaches seem to make it into an entirely separate language. In this introductory article, we discussed Clojure's approach to state and variables.

Here are some other Manning titles you might be interested in:

Scala in Action
Nilanjan Raychaudhuri



AspectJ in Action, Second Edition
Ramnivas Laddad

DSLs in Action
Debasish Ghosh

Last updated: August 19, 2011