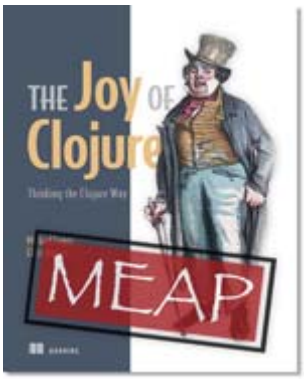
MANNING PUBLICATIONS

OOP in Clojure

Excerpted from



The Joy of Clojure

EARLY ACCESS EDITION

Michael Fogus and Chris Houser

MEAP Release: January 2010
Softbound print: Fall 2010 | 300 pages
ISBN: 9781935182641

This article is taken from the book The Joy of Clojure. *This section identifies the shortcomings of object-oriented programming and shows how Clojure, in most cases, improves those techniques.*

Most of what OOP gives you, Clojure provides

It's important to identify the shortcomings of object-oriented programming in order to improve the state (no pun intended) of concurrent projects. It's also important to realize which object-oriented features are powerful independent of concurrency. In this short discussion we'll identify a few of these powerful techniques and explain how Clojure adopts them. We won't go into extensive detail about these techniques now, but we'll identify them and go into further detail in our book, *The Joy of Clojure*.

Polymorphism

Polymorphism refers to the ability of methods with differing argument signatures to be treated as a single entity. In the days of C programming, nothing intrinsically supported function polymorphism. The ways around this involved a pairing of types and their acceptance functions, naming schemes, or single functions taking void pointers and performing casts as needed. The truly adventurous would pass everything as a giant struct taking only the parts needed, and sometimes others would devise their own unique version

of vtables. In any case, things could get hairy quickly. But ad hoc polymorphism introduced a way to dispatch a specific method from a set of identically named methods in a class or hierarchy based on the types of the parameters passed. Clojure, too, provides polymorphism in the form of multimethods and protocols. We won't delve into them here because they're covered more extensively later in the book. But we will say that multimethods in Clojure take the notion of polymorphism to the next logical step by allowing dispatch to occur based on the results of an arbitrary function.

Subtyping

Clojure provides a form of subtyping, albeit limited in scope compared to that found in object-oriented languages. For example, using the function `derive`, you can create ad hoc hierarchies in your own programs. Conversely, you can query these established hierarchies for their structure aspects using the functions `parents`, `ancestors`, `descendants`, and `isa?`. We'll delve into leveraging the ad hoc hierarchy facility in section 8.2, "Exploring multimethods with the Universal Design Pattern."

But what about encapsulation?

Imagine that we need a simple function that, given a representation of a chessboard and a coordinate, returns a simple representation of the piece at the given square. To keep matters as simple as possible, we'll use a vector containing a set of characters corresponding to the colored chess pieces, as shown in figure 1.

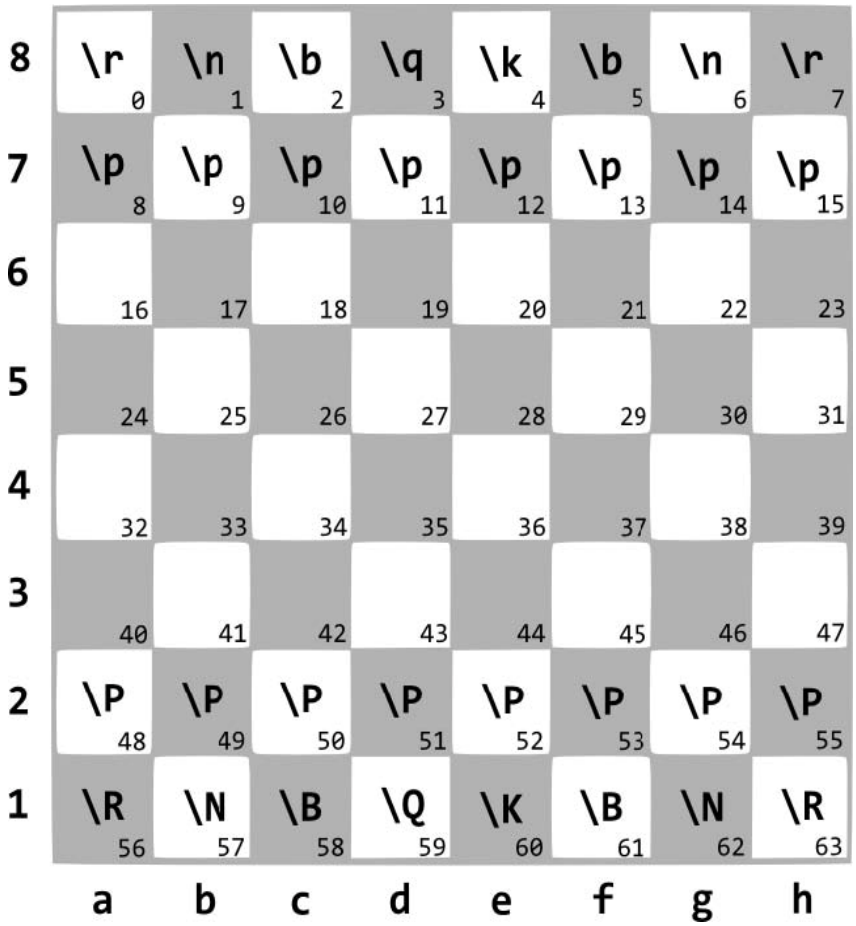


Figure 1 Chessboard diagram

There's no need to complicate matters with our chessboard representation; chess is hard enough. The data structure in the following code corresponds directly to a chessboard in the starting position:

```
(ns chess)

(defn initial-board []
  [\r \n \b \q \k \b \n \r
   \p \p \p \p \p \p \p \p   ; lowercase are dark
   \- \- \- \- \- \- \- \-
   \- \- \- \- \- \- \- \-
   \- \- \- \- \- \- \- \-
   \- \- \- \- \- \- \- \-
```

```
    \P \P \P \P \P \P \P \P   ; uppercase are light
    \R \N \B \Q \K \B \N \R])
```

From this we can gather that the black pieces are lowercase characters and the white pieces are uppercase. This kind of structure is likely not optimal, but it's a good start. We can ignore the implementation details for now and focus on the client interface to query the board for square occupations. This is a perfect opportunity to enforce encapsulation to avoid drowning the client in board-implementation details. Thankfully, programming languages with closures automatically support a form of encapsulation to group functions with their supporting data.[1]

```
;; Querying the squares of a chess board
(def *file-key* \a)
(def *rank-key* \0)

(defn- file-component
  "Calculate the file (horizontal) projection"
  [file]
  (- (int file) (int *file-key*)))

(defn- rank-component
  "Calculate the rank (vertical) projection"
  [rank]
  (* 8 (- 8 (- (int rank) (int *rank-key*)))))

(defn- index
  "Projecting the 1D layout onto a logical 2D chessboard"
  [file rank]
  (+ (file-component file) (rank-component rank)))

(defn lookup [board pos]
  (let [[file rank] pos]
    (board (index file rank))))
```

These functions are self-evident in their intent and are encapsulated at the level of the namespace. Namespace encapsulation is the most prevalent form of encapsulation that you'll encounter when exploring idiomatic source code. But the use of lexical closures provides more options for encapsulation: block-level and local encapsulation.

```
;; Using block-level encapsulation
(letfn [(index [file rank]
          (let [f (- (int file) (int \a))
                r (* 8 (- 8 (- (int rank) (int \0))))]
            (+ f r)))]
  (defn lookup [board pos]
```

---

1 In *JavaScript: The Good Parts* by Douglas Crockford, this form of encapsulation is described as the module pattern. The module pattern as implemented with JavaScript provides some level of data hiding also, whereas in Clojure, not so much.

```
    (let [[file rank] pos]
      (board (index file rank)))))
```

It's often a good idea to aggregate relevant data, functions, and macros at their most specific scope. In this section of code, we've taken the `file-component` and `rank-component` functions and the `*file-key*` and `*rank-key*` Vars out of the namespace proper and rolled them into a block-level index function defined with the body of the `letfn` macro. Within this body we then define the `lookup` function, limiting the client exposure to the chessboard API and hiding the implementation-specific functions and forms.[2] But we can further limit the scope of the encapsulation by shrinking the scope even more.

```
;; Local encapsulation
(defn lookup2 [board pos]
  (let [[file rank] (map int pos)
        [fc rc]     (map int [\a \0])
        f (- file fc)
        r (* 8 (- 8 (- rank rc)))
        index (+ f r)]
    (board index)))
```

We've now pulled *all* of the implementation-specific details into the body of the `lookup2` function itself. This localizes the scope of the `index` function and all auxiliary values to only the relevant party, namely, `lookup2`. As a nice bonus, `lookup2` is simple and compact without sacrificing readability. But bear in mind that this example is only illustrative of grouping encapsulation and not meant to provide a template for data-hiding encapsulation. Don't make the mistake of believing that your trade secrets are securely hidden within closures.

Everything isn't an object

Finally, another downside to object-oriented programming is the tight coupling methods and classes. Clojure functions are data, yet this in no way restricts the decoupling of data and the functions that work on them. Much of what programmers perceive to be classes are data tables that Clojure provides via maps and types. The final strike against viewing everything as an object is that mathematicians view little (if anything) as objects. Instead, mathematics is built on the relationships between one set of elements to another through the application of functions—*functional programming*.[3]

---

[2] The details are still easily accessible, but one has to intentionally jump a small fence to do so. We leave it as an exercise to the reader to find this fence.

[3] For a simple book to learn about functional programming as the application of functions to data, read *Common Lisp: A Gentle Introduction to Symbolic Computation* by David S. Touretzky.