SAMPLE CHAPTER

Clojure IN ACTION

SECOND EDITION

Amit Rathore
Francis Avila

MANNING

Clojure in Action
Second Edition

by Amit Rathore
Francis Avila

Chapter 8

brief contents

8 More on functional programming

This chapter covers

- A refresher on higher-order functions
- Partial application of functions
- Lexical closures
- Traditional object-oriented programming (OOP) in Clojure

So far you've seen a lot of the Clojure programming language, and you've used it to write a good number of functions. Because it's a functional programming language, understanding and mastering the functional programming paradigm is key to being successful with it. In this chapter, we'll explore this topic some more.

Instead of approaching this from, say, a mathematical (or plain theoretical) standpoint, we'll review code to explore some of the main ideas. We'll start by implementing a few common higher-order functions that you often see used in functional programs. The idea is to help you become comfortable with recursion, lazy sequences, functional abstraction, and function reuse.

Next, we'll visit the land of partial application. This exposure will give you further insight into functional programming and what you can do with it. Although

partial application doesn't find particularly widespread use in Clojure, sometimes it's the perfect fit for the job. Incidentally, you'll also see the alternatives to partial application later in the chapter.

The final stop will be to explore closures. The last section puts everything together to write a little object system that illustrates the ideas of OOP vis-à-vis functional programming. You may wonder why there's a section on OOP in a chapter on functional programming. Once you read through to the end of this chapter, you'll realize how functional programming can be thought of as a superset of OOP—and in fact transcends it.

8.1 Using higher-order functions

We talked about higher-order functions in chapter 3. A *higher-order function* is one that either accepts another function or returns a function. Higher-order functions allow the programmer to abstract out patterns of computation that would otherwise result in duplicated code. In this section, we'll look at a few examples of higher-order functions that can greatly simplify many things you'll come across. You've seen several of these functions before, in other forms, and we'll point these out as you implement them.

Overall, this section will give you a sense of how higher-order functions can be used to implement a variety of solutions in Clojure; indeed, how it's an integral part of doing so.

8.1.1 Collecting results of functions

Let's begin our look at higher-order functions by considering the idea of a function named `square-all` that accepts a list of numbers and returns a list of the squares of each element. You may need such a sequence in a graphics program or in some other math computation:

```
(defn square [x]
  (* x x))
(defn square-all [numbers]
  (if (empty? numbers)
    nil
    (cons (square (first numbers))
          (square-all (rest numbers)))))
```

Idiomatic equivalent to empty list

A quick note about returning `nil` in the empty case: you could also return an empty list, but in Clojure it's idiomatic to return `nil` instead because it's falsey (unlike an empty list, which is truthy) and all `seq`-related functions treat `nil` like an empty list anyway (for example, `conj`-ing onto `nil` returns a list).

Notice that the `cons` function is used to build a new sequence. It accepts a single element, and another sequence, and returns a new one with that element inserted in the first position. Here, the first element is the square of the first number, and the

body is the sequence of the squares of the remaining numbers. This works as expected, and you can test this at the read-evaluate-print loop (REPL) as follows:

```
(square-all [1 2 3 4 5 6])
;=> (1 4 9 16 25 36)
```

Now let's look at another function, `cube-all`, which also accepts a list of numbers but returns a list of cubes of each element:

```
(defn cube [x]
     (* x x x))
(defn cube-all [numbers]
  (if (empty? numbers)
    ()
    (cons (cube (first numbers))
          (cube-all (rest numbers)))))
```

Again, this is easy to test:

```
(cube-all [1 2 3 4 5 6])
;=> (1 8 27 64 125 216)
```

They both work as expected. The trouble is that there's a significant amount of duplication in the definitions of `square-all` and `cube-all`. You can easily see this commonality by considering the fact that both functions are applying a function to each input element and are collecting the results before returning the list of collected values.

You've already seen that such functions can be captured as higher-order functions in languages such as Clojure:

```
(defn do-to-all [f numbers]
  (if (empty? numbers)
    ()
    (cons (f (first numbers))
          (do-to-all f (rest numbers)))))
```

With this, you can perform the same operations easily:

```
(do-to-all square [1 2 3 4 5 6])
;=> (1 4 9 16 25 36)
(do-to-all cube [1 2 3 4 5 6])
;=> (1 8 27 64 125 216)
```

You can imagine that the `do-to-all` implementation is similar to that of the `map` function that's included in Clojure's core library. You've seen this function earlier in the book. The `map` function is an abstraction that allows you to apply any function across sequences of arguments and collect results into another sequence. This implementation is quite limited when compared with the core `map` function, and it also suffers

from a rather fatal flaw: it will blow the call stack if a long enough list of elements is passed in. Here's what it will look like:

```
(do-to-all square (range 11000))
StackOverflowError   clojure.lang.Numbers$LongOps.multiply (Numbers.java:459)
```

This is because every item results in another recursive call to `do-to-all`, thus adding another stack frame until you eventually run out. Consider the following revised implementation:

```
(defn do-to-all [f numbers]
  (lazy-seq
    (if (empty? numbers)
      ()
      (cons (f (first numbers))
            (do-to-all f (rest numbers))))))
```

Now, because this return is a lazy sequence, it no longer attempts to recursively compute all the elements to return. The `lazy-seq` macro takes a body of code that returns a sequence (or `nil`) and returns an object that's "seqable"—that is, it behaves like a sequence. But it invokes the body only once and on demand (lazily) and returns cached results thereafter. The function now works as expected:

```
(take 10 (drop 10000 (do-to-all square (range 11000))))
;=> (100000000 100020001 100040004 100060009 100080016 100100025 100120036
     100140049 100160064 100180081)
```

This is similar to the `map` function that comes with Clojure (although the Clojure version does a lot more). Here's what that might look like:

```
(take 10 (drop 10000 (do-to-all square (range 11000))))
;=> (100000000 100020001 100040004 100060009 100080016 100100025 100120036
     100140049 100160064 100180081)
```

Notice all that was done here was to replace `do-to-all` with `map`. The `map` function is an extremely useful higher-order function, and as you've seen over the last few chapters, it sees heavy use.

Let's now look at another important operation, which can be implemented using a different higher-order function.

8.1.2 Reducing lists of things

It's often useful to take a list of things and compute a value based on all of them. An example might be totaling a list of numbers or finding the largest number. You'll implement the total first:

```
(defn total-of [numbers]
  (loop [nums numbers sum 0]
    (if (empty? nums)
      sum
      (recur (rest nums) (+ sum (first nums))))))
```

This works as expected, as you can see in the following test at the REPL:

```
(total-of [5 7 9 3 4 1 2 8])
;=> 39
```

Now you'll write a function to return the greatest from a list of numbers. First, write a simple function that returns the greater of two numbers:

```
(defn larger-of [x y]
  (if (> x y) x y))
```

This is a simple enough function, but now you can use it to search for the largest number in a series of numbers:

```
(defn largest-of [numbers]
  (loop [l numbers candidate (first numbers)]
    (if (empty? l)
      candidate
      (recur (rest l) (larger-of candidate (first l))))))
```

You need to see if this works:

```
(largest-of [5 7 9 3 4 1 2 8])
;=> 9
(largest-of [])
;=> nil
```

It's working, but there's clearly some duplication in `total-of` and `largest-of`. Specifically, the only difference between them is that one adds an element to an accumulator, whereas the other compares an element with a candidate for the result. Next, you'll extract the commonality into a function:

```
(defn compute-across [func elements value]
  (if (empty? elements)
    value
    (recur func (rest elements) (func value (first elements)))))
```

Now you can easily use `compute-across` to implement `total-of` and `largest-of`:

```
(defn total-of [numbers]
  (compute-across + numbers 0))
(defn largest-of [numbers]
  (compute-across larger-of numbers (first numbers)))
```

To ensure that things still work as expected, you can test these two functions at the REPL again:

```
(total-of [5 7 9 3 4 1 2 8])
;=> 39
(largest-of [5 7 9 3 4 1 2 8])
;=> 9
```

`compute-across` is generic enough that it can operate on any sequence. For instance, here's a function that collects all numbers greater than some specified threshold:

```
(defn all-greater-than [threshold numbers]
  (compute-across #(if (> %2 threshold) (conj %1 %2) %1) numbers []))
```

Before getting into how this works, you need to check if it works:

```
(all-greater-than 5 [5 7 9 3 4 1 2 8])
;=> [7 9 8]
```

It works as expected. The implementation is simple: you've already seen how `compute-across` works. The initial value (which behaves as an accumulator) is an empty vector. You need to conjoin numbers to this when it's greater than the threshold. The anonymous function does this.

The `compute-across` function is similar to something you've already seen: the `reduce` function that's part of Clojure's core functions. Here's `all-greater-than` rewritten using the built-in `reduce`:

```
(defn all-greater-than [threshold numbers]
  (reduce #(if (> %2 threshold) (conj %1 %2) %1) [] numbers))
```

And here it is in action:

```
(all-greater-than 5 [5 7 9 3 4 1 2 8])
;=> [7 9 8]
```

Both the `compute-across` and `reduce` functions allow you to process sequences of data and compute a final result. Let's now look at another related example of using `compute-across`.

8.1.3 Filtering lists of things

You wrote a function in the previous section that allows you to collect all numbers greater than a particular threshold. Now you'll write another one that collects those numbers that are less than a threshold:

```
(defn all-lesser-than [threshold numbers]
  (compute-across #(if (< %2 threshold) (conj %1 %2) %1) numbers []))
```

Here's the new function in action:

```
(all-lesser-than 5 [5 7 9 3 4 1 2 8])
;=> [3 4 1 2]
```

Notice how easy it is, now that you have your convenient little `compute-across` function (or the equivalent `reduce`). Also, notice that there's duplication in the `all-greater-than` and `all-lesser-than` functions. The only difference between them is in the criteria used in selecting which elements should be returned.

Now you need to extract the common part into a higher-order `select-if` function:

```
(defn select-if [pred elements]
  (compute-across #(if (pred %2) (conj %1 %2) %1) elements []))
```

You can now use this to select all sorts of elements from a larger sequence. For instance, here's an example of selecting all odd numbers from a vector:

```
(select-if odd? [5 7 9 3 4 1 2 8])
;=> [5 7 9 3 1]
```

To reimplement the previously defined `all-lesser-than` function, you could write it in the following manner:

```
(defn all-lesser-than [threshold numbers]
  (select-if #(< % threshold) numbers))
```

This implementation is far more readable, because it expresses the intent with simplicity and clarity. The `select-if` function is another useful, low-level function that you can use with any sequence. In fact, Clojure comes with such a function, one that you've seen before: `filter`. Here, for instance, is the same selection of odd numbers you just saw:

```
(filter odd?  [5 7 9 3 4 1 2 8])
;=> (5 7 9 3 1)
```

Note that although `filter` returns a lazy sequence here, and the `select-if` function returned a vector, as long as your program expects a sequence, either function will work.

Over the last few pages, you've created the functions `do-to-all`, `compute-across`, and `select-if`, which implement the essence of the built-in `map`, `reduce`, and `filter` functions. The reason for this was two-fold: to demonstrate common use cases of higher-order functions and to show that the basic form of these functions is rather simple to implement. The `select-if` isn't lazy, for instance, but with all the knowledge you've gained so far, you can implement one that is. With this background in place, let's explore a few other topics of interest of functional programs.

8.2 Partial application

In the last section you wrote several higher-order functions that accepted a function as one argument and applied it to other arguments. Now we're going to look at another kind of higher-order function—those that create and return new functions. This is a crucial aspect of functional programming, and in this section you'll write functions that return new functions of less arity than the ones they accept as an argument. You'll do this by "partially applying" the function. (Don't worry; the meaning of this will become clear shortly.)

8.2.1 Adapting functions

Let's imagine that you have a function that accepts a tax percentage (such as 8.0 or 9.75) and a retail price and returns the total price with tax using a threading macro:

```
(defn price-with-tax [tax-rate amount]
  (->> (/ tax-rate 100)
       (+ 1)
       (* amount)))
```

Now you can find out what something truly costs, because you can calculate its price including the sales tax, as follows:

```
(price-with-tax 9.5M 100)
;=> 109.500M
```

Notice that you use the `BigDecimal` type for money (specified by the `M` suffix): using floating-point numbers for financial applications is asking for rounding errors! If you had a list of prices that you wanted to convert into a list of tax-inclusive prices, you could write the following function:

```
(defn with-california-taxes [prices]
  (map #(price-with-tax 9.25M %) prices))
```

And you could then batch-calculate pricing with taxes:

```
(def prices [100 200 300 400 500])
;=> #'user/prices
(with-california-taxes prices)
;=> (109.2500M 218.5000M 327.7500M 437.0000M 546.2500M)
```

Notice that in the definition of `with-california-taxes`, there's an anonymous function that accepted a single argument (a price) and applied `price-with-tax` to 9.25 and the price. Creating this anonymous function is convenient; otherwise, you might have had to define a separate function that you may never have used anywhere else, such as this:

```
(defn price-with-ca-tax [price]
  (price-with-tax 9.25M price))
```

And if you had to handle New York, it would look like this:

```
(defn price-with-ny-tax [price]
  (price-with-tax 8.0M price))
```

If you had to handle any more, the duplication would certainly get to you. Luckily, a functional language such as Clojure can make short work of it:

```
(defn price-calculator-for-tax [state-tax]
  (fn [price]
    (price-with-tax state-tax price)))
```

This function accepts a tax rate, presumably for a given state, and then returns a new function that accepts a single argument. When this new function is called with a price, it returns the result of applying `price-with-tax` to the originally supplied tax rate and the price. In this manner, the newly defined (and returned) function behaves like a closure around the supplied tax rate. Now that you have this higher-level function, you can remove the duplication you saw earlier by defining state-specific functions as follows:

```
(def price-with-ca-tax (price-calculator-for-tax 9.25M))
(def price-with-ny-tax (price-calculator-for-tax 8.0M))
```

Figure 8.1 shows how this works.

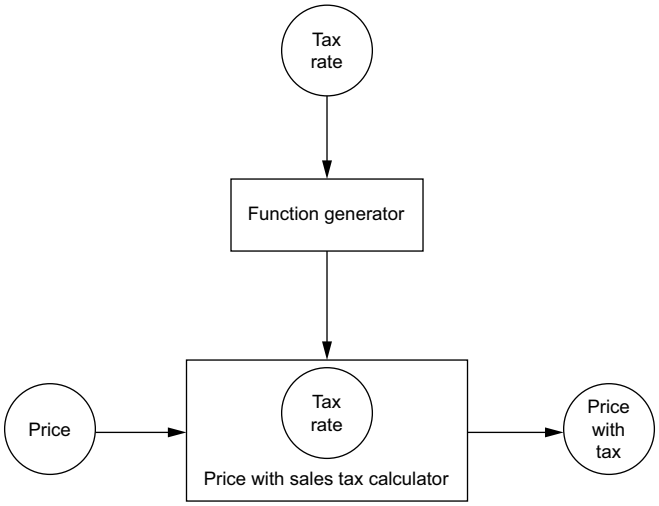

Figure 8.1 The `tax-rate` initially passed into the `price-calculator-for-tax` function is captured by the resulting function (such as `price-with-ca-tax`). It's then able to use that value when it's called with a price, to return the correct price with the tax applied to it.

Notice again that you're creating new vars (and that you're directly using `def` here, not `defn`) that are bound to the anonymous functions returned by the `price-calculator-for-tax` function.

These new functions accept a single argument and are perfect for functions such as `with-california-taxes` that accept a list of prices, and call `map` across them. A single-argument function serves well in such a case, and you can use any of the previous functions for this purpose. This is a simple case where you started out with a function of a certain arity (in this case, `price-with-tax` accepts two arguments), and you needed a new function that accepted a lesser number of arguments (in this case, a single-argument function that could map across a sequence of prices).

This approach of taking a function of n arguments and creating a new function of k arguments (where $n > k$) is a form of adaptation (you may be familiar with the adapter pattern from OOP literature). You don't need special ceremony to do this in functional languages, thanks to first-class functions. Let's see how Clojure makes this easy.

PARTIAL APPLICATION

Let's say you have a function of n arguments and you need to fix $(n - k)$ arguments to create a new function of k arguments. Here's a function to illustrate this:

```
(defn of-n-args [a b c d e]
  (str a b c d e ))
```

Now, to fix, say, the first three arguments to 1, 2, and 3, you could do the following:

```
(defn of-k-args [d e]
  (of-n-args 1 2 3 d e))
```

You need to ensure that this function works as expected:

```
(of-k-args \a \b)
;=> "123ab"
```

Okay, so that works. If you needed to create a function that fixed, say, two or four arguments, you'd have to write similar code again. As you can imagine, if you had to do this a lot, it would get rather repetitive and tedious.

You could improve things by writing a function that generalizes the idea, such as

```
(defn partially-applied [of-n-args & n-minus-k-args]
  (fn [& k-args]
    (apply of-n-args (concat n-minus-k-args k-args))))
```

Now, you could create any number of functions that fixed a particular set of arguments of a particular function, for example:

```
(def of-2-args (partially-applied of-n-args \a \b \c))
;=> #'user/of-2-args
(def of-3-args (partially-applied of-n-args \a \b))
;=> #'user/of-3-args
```

And you can see if these work as expected:

```
(of-2-args 4 5)
;=> "abc45"
(of-3-args 3 4 5)
;=> "ab345"
```

The new function is called `partially-applied` because it returns a function that's a partially applied version of the function you passed into it. For example, `of-3-args` is a partially applied version of `of-n-args`. This is such a common technique in functional programming that Clojure comes with a function that does this, and it's called `partial`.

It's used the same way:

```
(def of-2-args (partial of-n-args \a \b \c))
;=> #'user/of-2-args
(def of-3-args (partial of-n-args \a \b))
;=> #'user/of-3-args
```

And here it is in action:

```
(of-2-args 4 5)
;=> "abc45"
(of-3-args 3 4 5)
;=> "ab345"
```

You now understand what it means to partially apply a function. Partial application is an abstraction that comes out of having higher-order functions. Although the examples showed this technique where you needed to adapt a function of a given arity to a function of a lower arity, there are other uses as well. You'll see one such use in the next section.

8.2.2 Defining functions

In this section, we'll use the technique of partial application to define new functions. Recall the `select-if` function from the previous section:

```
(defn select-if [pred elements]
  (compute-across #(if (pred %2) (conj %1 %2) %1) elements []))
```

Note that the `compute-across` function is passed an empty vector as the last argument. Here, you'll write a modified version of `select-if` called `select-into-if`, which will accept an initial container:

```
(defn select-into-if [container pred elements]
  (compute-across #(if (pred %2) (conj %1 %2) %1) elements container))
```

Again, as you saw in the previous section, if you had a list of numbers such as

```
(def numbers [4 9 5 7 6 3 8])
```

then you could use the new function as follows:

```
(select-into-if [] #(< % 7) numbers)
;=> [4 5 6 3]
```

Similarly, you could also pass in an empty list instead of an empty vector, as shown here:

```
(select-into-if () #(< % 7) numbers)
;=> (3 6 5 4)
```

Note that depending on whether you want to filter results up or down (in the same order as the elements appear or in the reverse), you can use either the empty vector as your container or the empty list. This ordering may be useful when you have a situation where the order matters, and you either want to preserve it or perhaps reverse it. Now you'll further abstract this idea of filtering results up or down as follows:

```
(def select-up (partial select-into-if []))
```

This is a new function using `partial`. You fixed the first argument of the `select-into-if` function to the empty vector. Similarly, you could define the concept of selecting down a sequence of elements as follows:

```
(def select-down (partial select-into-if ()))
```

It's time to test these two functions to ensure that they work:

```
(select-up  #(< % 9) [5 3 9 6 8])
;=> [5 3 6 8]
(select-down  #(< % 9) [5 3 9 6 8])
;=> (8 6 3 5)
```

Obviously, there are specific implications that arise from using a vector versus a list, and depending on the situation, these may be a convenient way to filter elements of a sequence.

As you've seen, partial application of functions can be a useful tool. This section showed two situations where this technique might come in handy. The first is to adapt functions to a suitable arity by fixing one or more arguments of a given function. The second is to define functions by partially applying a more general function to get specific functions that have one or more arguments fixed.

There's a more general principle at work here, a principle that makes partial function application possible: lexical closures.

8.3 Closures

In this section, we're going to explore the lexical closure, which is a central concept of functional programming. How central it is can be seen from the fact that the name *Clojure* itself is a play on the term *closure*. We'll begin by reviewing what closures are and what they close over. Then we'll look at a couple of closure use cases in functional programming. Specifically, we'll look at how closures can be used to delay computation and how they can be used as objects (in the OO sense of the word). At the end, we'll create a little OO layer for Clojure to demonstrate that Clojure (and Lisps in general) go beyond traditional ideas of OOP.

8.3.1 Free variables and closures

Before we jump into what closures are, let's look at the concept of free variables. A free variable is one that's neither an argument nor a local variable. For instance, take a look at `adder` in this code:

```
(defn adder [num1 num2]
  (let [x (+ num1 num2)]
    (fn [y]
      (+ x y))))
```

`x` is a free variable: its value will come from outside scope of function that uses it.

Here, `num1` and `num2` are arguments of the `adder` function and so they aren't free variables. The `let` form creates a lexically scoped block of code, and within that block,

num1 and num2 are free variables. Further, the let form creates a locally named value called x. Therefore, within the let block, x isn't a free variable. Finally, an anonymous function is created that accepts an argument called y. Within this anonymous function, y isn't a free variable, but x is (because it's neither an argument nor a local variable within the function block).

Now that you understand what a free variable is, let's examine this function a little more. Consider the following code:

```
(def add-5 (adder 2 3))
;=> #'user/add-5
```

add-5 is a var that's bound to the return value of the call to adder, which is the anonymous function returned by the adder function. The function object contains within it a reference to x, which exists for only as long as the let block inside adder lives. Consider that the following works as expected:

```
(add-5 10)
;=> 15
```

Given the fact that the life of a locally named value such as x lasts only until the enclosing lexical block lasts, how can add-5 do its work? You might imagine that the x referenced inside add-5 ceased to be the moment add-5 was created.

The reason this works, though, is that the anonymous function returned by adder is a closure, in this case closing over the free variable x. The extent (life) of such closed-over free variables is that of the closure itself. This is why add-5 is able to use the value of x, and it adds 10 to 5 to return 15 in the example.

In summary, a free variable will have the value that variable has in its enclosing scope at the moment that its own scope is created. Now let's look at what closures can do for your Clojure programs.

8.3.2 Delayed computation and closures

One aspect of closures is that they can be executed at any time, any number of times, or not at all. This property of delayed execution can be useful. Consider the following code:

```
(let [x 1
      y 0]
  (/ x y))
ArithmeticException Divide by zero  clojure.lang.Numbers.divide
(Numbers.java:156)
```

You know what this code will do: it will promptly throw a nice divide-by-zero exception. Let's wrap the code with a try-catch block so as to control the situation programmatically:

```
(let [x 1
      y 0]
    (try
      (/ x y)
      (catch Exception e (println (.getMessage e)))))
Divide by zero
;=> nil
```

This pattern is common enough that you might want to extract it out into a higher-order control structure:

```
(defn try-catch [the-try the-catch]
  (try
    (the-try)
    (catch Exception e (the-catch e))))
```

Now that you have this, you could write

```
(let [x 1
      y 0]
  (try-catch #(/ x y)
             #(println (.getMessage %))))
```

Notice here that you're passing in an anonymous function that closes around `x` and `y`. You could have written a macro to do this same thing, but this shows that you don't need macros for something like this, even though the macro solution would be nicer (syntactically more convenient to use, resulting in code that's easier to read).

When we compared macros to functions, we noted that Clojure evaluates function arguments in an eager manner. This behavior is one of the reasons why macros have an advantage, in that macros don't evaluate arguments, allowing us as programmers to be in control. Here, the `try-catch` function achieves the same intended effect by accepting functions that will be evaluated later. To be specific, although your anonymous functions are created immediately (when you passed them in as arguments), they're only *evaluated* later (within the `try-catch` block). Further, because the free variables `x` and `y` were enclosed within a closure correctly, the `try-catch` function was able to work correctly.

You can imagine creating other control structures in a similar manner. In the next section, you'll see another interesting aspect of closures.

8.3.3 Closures and objects

In this section, we're going to examine another benefit of the closure. As you've seen over the past few paragraphs, a closure captures the bindings of any free variables visible at the point of creation. These bindings are then hidden from view to the rest of the world, making the closure a candidate for private data. (Data hiding is somewhat of an overrated concept, especially in a language such as Clojure. You'll see more on this in the next section.)

For the moment, let's continue exploring the captured bindings inside closures. Imagine that you needed to handle users' login information and email addresses in your application. Consider the following function:

```
(defn new-user [login password email]
  (fn [a]
    (case a
      :login login
      :password password
      :email email
      nil)))
```

There's nothing particularly new here; you saw such code in the previous section. Here it is in action:

```
(def arjun (new-user "arjun" "secret" "arjun@zololabs.com"))
;=> #'user/arjun
(arjun :login)
;=> "arjun"
(arjun :password)
;=> "secret"
(arjun :email)
;=> "arjun@zololabs.com"
(arjun :name)
;=> nil
```

First, a new function object is created by calling the `new-user` function with `"arjun"`, `"secret"`, and `"arjun@currylogic.com"` as arguments. Then, you were able to query it for the login, password, and email address. `arjun` appears to behave something like a map, or at least a data object of some kind. You can query the internal state of the object using the keywords `:login`, `:password`, and `:email`, as shown previously.

It's also worth noting that this is the only way to access the internals of `arjun`. This is a form of message passing: the keywords are messages that you're sending to a receiver object, which in this case is the function object `arjun`. You can implement fewer of these should you choose to. For instance, you might deem the password to be a hidden detail. The modified function might look as follows:

```
(defn new-user [login password email]
  (fn [a]
    (case a
      :login login
      :email email
      :password-hash (hash password)
      nil)))
;=> #'user/new-user
(def arjun (new-user "arjun" "secret" "arjun@zololabs.com"))
;=> #'user/arjun
(arjun :password)
;=> nil
(arjun :password-hash)
;=> 1614358358
```

Inner function can access `password`.

Doesn't ever return password to callers

Enables information hiding; can use `password` without letting anyone else see it

Although the `new-user` function and the returned inner function can see `password`, anyone using the returned inner function can only see the hash of the password. Using a closure, you were able to hide information from callers of a function that's still visible to the function itself. Having come this far, we'll take a short break to compare functions such as `arjun` with objects from other languages.

DATA OR FUNCTION?

Already the line between what's clearly a function and what might be construed to be data in languages such as Java and Ruby should have started to blur. Is `arjun` a function or is it a kind of data object? It certainly behaves similarly to a hash map, where you can query the value associated with a particular key. In this case, you wrote code to expose only those keys that you considered public information, while hiding the private pieces. Because `arjun` is a closure, the free variables (the arguments passed to `new-user`) are captured inside it and hang around until the closure itself is alive. Technically, although `arjun` is a function, semantically it looks and behaves like data.

Although data objects such as hash maps are fairly static (in that they do little more than hold information), traditional objects also have behavior associated with them. Let's blur the line some more by adding behavior to your user objects. Here you'll add a way to see if a given password is the correct one. Consider this code:

```
(defn new-user [login password email]
  (fn [a & args]
    (case a
      :login login
      :email email
      :authenticate (= password (first args)))))
```

Now try it out:

```
(def adi (new-user "adi" "secret" "adi@currylogic.com"))
;=> #'user/adi
(adi :authenticate "blah")
;=> false
(adi :authenticate "secret")
;=> true
```

Your little closure-based users can now authenticate themselves when asked to do so. As mentioned earlier, this form of message passing resembles calling methods on objects in languages such as Java and Ruby. The format for doing so could be written like so:

```
(object message-name & arguments)
```

Objects in OOP are usually defined as entities that have state, behavior, and equality. Most languages also allow them to be defined in a manner that allows inheritance of functionality. So far, we've handled state (information such as `login`, `password`, and

email) and behavior (such as :authenticate). Equality depends on the domain of use of these objects, but you could conceivably create a generic form of equality testing based on, say, a hash function. In the next section, we'll consolidate the ideas we talked about in this section and add inheritance and a nicer syntax to define such objects.

8.3.4 An object system for Clojure

In the previous section, you created a function named new-user that behaves as a sort of factory for new user objects. You could call new-user using the data elements that a user comprises (login, password, and email address), and you'd get a new user object. You could then query it for certain data or have it perform certain behaviors. You made this possible by implementing a simple message passing scheme, with messages such as :login and :authenticate.

In this section, we'll generalize the idea of creating objects that have certain data and behavior into what traditional OO languages call classes. First, you'll allow a simple class hierarchy, and you'll let objects refer to themselves by providing a special symbol, traditionally named this. Finally, you'll wrap this functionality in a syntactic skin to make it appear more familiar.

DEFINING CLASSES

We'll start simple. You'll create the ability to define classes that have no state or behavior. You'll lay the foundations of your object system starting with the ability to define a class that's empty. For instance, you'd like to be able to say

```
(defclass Person)
```

This would define a class that would behave as a blueprint for future instances of people. You could then ask for the name of such a class:

```
(Person :name)
;=> "Person"
```

That would return the string "Person", which is the name of the class you're defining. Once you implement this, you can start adding more functionality to your little object system. Consider the following implementation:

```
(defn new-class [class-name]
  (fn [command & args]
    (case command
      :name (name class-name))))
(defmacro defclass [class-name]
  `(def ~class-name (new-class '~class-name)))
```

So your little Person class is a function that closes over the class-name, passed in along with a call to defclass. Again, remember that the notation '~class-name quotes

the value of the `class-name` argument, so that it's passed in as the argument to `new-class` as a symbol itself.

Your classes support a single message right now, which is `:name`. When passed `:name`, the `Person` function returns the string `"Person"`. Try the following at the REPL now:

```
(defclass Person)
;=> #'user/Person
(Person :name)
;=> "Person"
```

Further, because you're simply working with vars and functions, the following also works:

```
(def some-class Person)
;=> #'user/some-class
(some-class :name)
;=> "Person"
```

This is to show that the name of the class isn't associated with the var (in this case `Person`) but with the class object itself. Now that you can define classes, we'll make it so you can instantiate them.

CREATING INSTANCES

Because your classes are closures, you can also implement instantiated classes as closures. You'll create this with a function called `new-object` that will accept the class that you'd like to instantiate. Here's the implementation:

```
(defn new-object [klass]
  (fn [command & args]
    (case command
      :class klass)))
```

Here it is in action:

```
(def cindy (new-object Person))
```

As you can tell, the only message you can send this new object is one that queries its class. You can say

```
(new-object Person)
;=> #<user$new_object$fn__2259 user$new_object$fn__2259@1f106fec>
```

That isn't terribly informative, because it's returning the class as the function object. You can instead further ask its name:

```
((cindy :class) :name)
;=> "Person"
```

You could add this functionality as a convenience message that an instantiated object itself could handle:

```
(defn new-object [klass]
  (fn [command & args]
    (case command
      :class klass
      :class-name (klass :name))))
```

You can test this now, but you'll need to create the object again, because you redefined the class. Here it is:

```
(def cindy (new-object Person))
;=> #'user/cindy
(cindy :class-name)
;=> "Person"
```

Finally, you're going to be instantiating classes a lot, so you can add a more convenient way to do so. You'd like something like a `new` operator that's common to several languages such as Java and Ruby. Luckily, you've already set up your class as a function that can handle incoming messages, so you'll add to the vocabulary with a `:new` message. Here's the new implementation of `new-class`:

```
(defn new-class [class-name]
  (fn klass [command & args]
    (case command
      :name (name class-name)
      :new (new-object klass))))
```

Notice that `new-object` accepts a class, and you need to refer to the class object from within it. You were able to do this by giving the anonymous function a name (`klass`) and then refer to it by that name in the `:new` clause. With this, creating new objects is easier and more familiar (remember to evaluate the definition of `Person` again):

```
(defclass Person)
;=> #'user/Person
(def nancy (Person :new))
;=> #'user/nancy
(nancy :class-name)
;=> "Person"
```

So here you are: you're able to define new classes as well as instantiate them.

Our next stop will be to allow your objects to maintain state.

OBJECTS AND STATE

In chapter 6 we explored Clojure's support for managing state. In this section, you'll use one of those available mechanisms to allow objects in your object system to also become stateful. You'll use a ref so that your objects will be able to participate in coordinated transactions. You'll also expand the vocabulary of messages that your objects

can understand by supporting `:set!` and `:get`. They'll allow you to set and fetch values from your objects, respectively.

Consider the following updated definition of the `new-object` function:

```
(defn new-object [klass]
  (let [state (ref {})]
    (fn [command & args]
      (case command
        :class klass
        :class-name (klass :name)
        :set! (let [[k v] args]
                (dosync (alter state assoc k v))
                nil)
        :get (let [[key] args]
               (@state key))))))
```

So now the messages that you can pass to your objects include `:class`, `:class-name`, `:set!`, and `:get`. Let's see your new stateful objects in action:

```
(def nancy (Person :new))
;=> #'user/nancy
(nancy :get :name)
;=> "Nancy Warhol"
```

You can also update your objects using the same `:set!` message, as follows:

```
(nancy :set! :name "Nancy Drew")
;=> nil
(nancy :get :name)
;=> "Nancy Drew"
```

You're coming along in the journey to create a simple object system. You now have the infrastructure to define classes, instantiate them, and manage state. Our next stop will be to add support for method definitions.

Defining methods

So far, we've dealt with the state side of objects. In this section, we'll start working on behavior by adding support for method definitions.

In keeping with languages such as Java and C++, you'd like to support a syntax that lists the methods along with the class definition, like so:

```
(defclass Person
  (method age []
    (* 2 10))
  (method greet [visitor]
    (str "Hello there, " visitor)))
```

You've started with simple methods. The `age` method, for instance, doesn't take any arguments and returns the result of a simple computation. Similarly, the `greet` method accepts a single argument and returns a simple computation involving it.

Now that the expected syntax is laid out, you can go about the implementation. First, you'll work on `defclass`, to make the previous notation valid. Consider the following function, which operates on a single method definition s-expression:

```
(defn method-spec [sexpr]
  (let [name (keyword (second sexpr))
    body (next sexpr)]
    [name (conj body 'fn)]))
```

This creates a vector containing the name of the method definition (as a keyword) and another s-expression, which can later be evaluated to create an anonymous function. Here's an example:

```
(method-spec '(method age [] (* 2 10)))
;=> [:age (fn age [] (* 2 10))]
```

Because you're going to specify more than one method inside the class definition, you'll need to call `method-spec` for each of them. The following `method-specs` function will accept the complete specification of your class, pick out the method definitions by filtering on the first symbol (it should be `method`), and then call `method-spec` on each:

```
(defn method-specs [sexprs]
  (->> sexprs
       (filter #(= 'method (first %)))
       (mapcat method-spec)
       (apply hash-map)))
```

The easiest way to see what's going on is to examine a sample output:

```
(method-specs '((method age [] (* 2 10))
                (method greet [visitor] (str "Hello there, " visitor))))
;=> {:age (fn age [] (* 2 10)),
     :greet (fn greet [visitor] (str "Hello there, " visitor))}
```

You now have a literal map that can be evaluated to return one containing keywords as keys for each method definition and an associated anonymous function. This map could then be passed to the `new-class` function for later use. Here's the associated revision of `new-class`:

```
(defn new-class [class-name methods]
  (fn klass [command & args]
    (case command
      :name (name class-name)
      :new (new-object klass))))
```

Now that all the supporting pieces are in place, you can make the final change to `defclass`, which will allow you to accept method definitions:

```
(defmacro defclass [class-name & specs]
  (let [fns (or (method-specs specs) {})]
    `(def ~class-name (new-class '~class-name ~fns))))
```

Now, your desired syntax from before will work:

```
(defclass Person
  (method age [] (* 2 10))
  (method greet [visitor] (str "Hello there, " visitor)))
;=> #'user/Person
```

So you've successfully made it possible to specify methods along with the class definitions. Note that the definition of `new-class` doesn't yet do anything with `methods`. Now, all you have to do is extend your objects so that you can invoke these methods.

INVOKING METHODS

To be able to invoke a method on one of your objects, such as `nancy`, you'll need some way to look up the definition from the associated class. You'll need to be able to query the class of a given object for a particular method. Let's add the `:method` message to your classes, which would accept the name of the method you're looking for. Consider the following revision to the `new-class` function:

```
(defn new-class [class-name methods]
  (fn klass [command & args]
    (case command
      :name (name class-name)
      :new (new-object klass)
      :method (let [[method-name] args]
                (find-method method-name methods)))))
```

We haven't defined `find-method` yet, so this code isn't quite ready to be compiled. To find a method from your previously created map of methods, you can do a simple hash map lookup. Therefore, the implementation of `find-method` is simple:

```
(defn find-method [method-name instance-methods]
  (instance-methods method-name))
```

With this addition, you can look up a method in a class, using the same keyword notation you've been using for all your other messages. Here's an example:

```
(Person :method :age)
;=> #<user$age user$age@42443032>
```

Now that you can get a handle on the function object that represents a method, you're ready to call it. Indeed, you can call the previous function on the REPL, and it will do what you laid out in the class definition:

```
((Person :method :age))
;=> 20
```

That works, but it's far from pretty. You should be able to support the same familiar interface of calling methods on objects that they belong to. Let's expand the capability

of the message-passing system you've built so far to handle such methods also. Here's an updated version of `new-object` that does this:

```
(defn new-object [klass]
  (let [state (ref {})]
    (fn [command & args]
      (case command
        :class klass
        :class-name (klass :name)
        :set! (let [[k v] args]
                (dosync (alter state assoc k v))
                nil)
        :get (let [[key] args]
               (@state key))
        (if-let [method (klass :method command)]
          (apply method args)
          (throw (RuntimeException.
                   (str "Unable to respond to " command)))))))))
```

What's added here is a default clause to `case`. If the message passed in isn't one of `:class`, `:class-name`, `:set!`, or `:get`, then you assume it's a method call on the object. You ask the class for the function by passing along the received command as the method name, and if you get back a function, you execute it. Here it is in action:

```
(def shelly (Person :new))
;=> #'user/shelly
(shelly :age)
;=> 20
(shelly :greet "Nancy")
;=> "Hello there, Nancy"
```

Remember, for this to work, the definition of `Person` would need to be reevaluated after these changes.

Once you're satisfied that your implementation does what you want so far, you'll be ready to move on to enabling objects to refer to themselves.

REFERRING TO THIS

So far, the method definitions have been simple. But there's often a need for methods within a class definition to call each other. Most programming languages that support this feature do so via a special keyword (usually named `this` or `self`), which refers to the object itself. Here you'll support the same functionality by providing a special name, which is also called `this`.

Once you've finished, you'd like to be able to say the following:

```
(defclass Person
  (method age [] (* 2 10))
  (method about [diff]
    (str "I was born about " (+ diff (this :age)) " years ago")))
```

Notice how the `about` method calls the `age` method via the `this` construct. To implement this, you'll first create a var named `this`, so that the class definitions continue to work (without complaining about unresolved symbols):

```
(declare ^:dynamic this)
```

This var will need a binding when any method executes so that its bound value refers to the object itself. A simple `binding` form will do, as long as you have something to bind to. You'll employ the same trick you did when you named the anonymous class function `klass` earlier, by naming the anonymous object function `thiz`. Here's the updated code for the `new-object` function:

```
(defn new-object [klass]
  (let [state (ref {})]
    (fn thiz [command & args]
      (case command
        :class klass
        :class-name (klass :name)
        :set! (let [[k v] args]
                (dosync (alter state assoc k v))
                nil)
        :get (let [[key] args] (@state key))
        (let [method (klass :method command)]
          (if-not method
            (throw (RuntimeException.
              (str "Unable to respond to " command))))
          (binding [this thiz]
            (apply method args)))))))
```

And that's all there is to it. Remember to evaluate the new, revised definition of `new-object` (and `Person`), and then you can confirm that it works on the REPL:

```
(def shelly (Person :new))
;=> #'user/shelly
(shelly :about 2)
;=> "I was born about 22 years ago"
```

You've added almost all the features you wanted to when we started this section. The last thing you'll add is the ability of a class to inherit from another.

CLASS INHERITANCE

You're about to add a final feature to your little object system. Traditional OOP languages such as Java and Ruby allow modeling of objects using inheritance so that problems can be decomposed into hierarchies of functionality. The lower in a hierarchy you go, the more specific you get. For instance, `Animal` might be a parent class of `Dog`. In this section, you'll add the ability to do that.

The first thing you'll do is allow the class definition to specify the parent class. Imagine that your syntax will look like this, with a choice of the word `extends` to signify the hierarchy:

```
(defclass Woman
  (extends Person)
  (method greet [v] (str "Hello, " v))
  (method age [] (* 2 9)))
```

Here, the new `Woman` class inherits from a previously defined `Person` class. The term `extends` is used to signify this relationship, as is common to other OO languages.

Now that you have your notation, you need to implement it. The first step is to write a function that can extract the parent class information from the class definition. Before you can do so, you have to decide what to do if a parent class isn't provided.

Again, you'll look at other languages for the answer. Your class hierarchies will all be singly rooted, and the top-level class (highest parent class) will be `OBJECT`. We'll define this shortly. For now, you're ready to write `parent-class-spec`, the function that will parse the specification of the parent class from a given class definition:

```
(defn parent-class-spec [sexprs]
  (let [extends-spec (filter #(= 'extends (first %)) sexprs)
        extends (first extends-spec)]
    (if (empty? extends)
      'OBJECT
      (last extends))))
```

To confirm that this works, try it at the REPL. You'll pass it the specification part of a class definition:

```
(parent-class-spec '((extends Person)
                     (method age [] (* 2 9))))
;=> Person
```

Now that you have the parent class, you'll pass it to the `new-class` function. You don't want to pass a symbol as the parent class but rather the `var` named by that symbol. For instance, the value returned by the call to `parent-class-spec` is a Clojure symbol. If you have a symbol, you can find the `var` named by the symbol using the `var` special form:

```
(var map)
;=> #'clojure.core/map
```

There's a reader macro for the `var` special form, `#'` (the hash followed by a tick). With this information in hand, you can make the needed modification to `defclass`:

```
(defmacro defclass [class-name & specs]
  (let [parent-class (parent-class-spec specs)
        fns (or (method-specs specs) {})]
    `(def ~class-name (new-class '~class-name  #'~parent-class ~fns))))
```

Note that you're now passing an extra parameter (the parent class) to the `new-class` function, so you'll have to change that to accommodate it:

```
(defn new-class [class-name parent methods]
  (fn klass [command & args]
    (case command
      :name (name class-name)
      :parent parent
      :new (new-object klass)
      :method (let [[method-name] args]
                (find-method method-name methods)))))
```

There's one more thing to handle before you can use your new `defclass`. You're looking up the parent class using the `var` special form, so you'll need `OBJECT` to resolve to something. It's now time to define it:

```
(def OBJECT (new-class :OBJECT nil {}))
```

With these changes, the definition of the `Woman` class from earlier in this section should work. Check this on the REPL as follows:

```
(defclass Person
       (method age [] (* 2 10))
       (method about [diff]
         (str "I was born about " (+ diff (this :age)) " years ago")))
;=> #'user/Person
```

This is your parent class; now you'll inherit from it to create the `Woman` class:

```
(defclass Woman
       (extends Person)
       (method greet [v] (str "Hello, " v))
       (method age [] (* 2 9)))
;=> #'user/Woman
```

You've only half-finished the job you started out to do. Although you can specify the parent class in your calls to `defclass`, method calls on your objects won't work right with respect to parent classes.

The following illustrates the problem:

```
(def donna (Woman :new))
;=> #'user/donna
(donna :greet "Shelly")
;=> "Hello, Shelly"
(donna :age)
;=> 18
(donna :about 3)
RuntimeException Unable to respond to :about  user/new-object/thiz--2733
(NO_SOURCE_PATH:1:1)
```

To fix this last error, you'll have to improve the method lookup to find the method in the parent class. Indeed, you'll have to search up the hierarchy of classes (parent of

the parent of the parent …) until you hit `OBJECT`. You'll implement this new method lookup by modifying the `find-method` function as follows:

```
(defn find-method [method-name klass]
  (or ((klass :methods) method-name)
      (if-not (= #'OBJECT klass)
        (find-method method-name (klass :parent)))))
```

For this to work, you'll need to have the classes handle another message, namely `:methods`. Also, the classes will use this new version of `find-method` to perform a method lookup. Here's the updated code:

```
(defn new-class [class-name parent methods]
  (fn klass [command & args]
    (case command
      :name (name class-name)
      :parent parent
      :new (new-object klass)
      :methods methods
      :method (let [[method-name] args]
                (find-method method-name klass)))))
```

With this final change, your object system will work as planned. Figure 8.2 shows the conceptual model of the class system you've built.

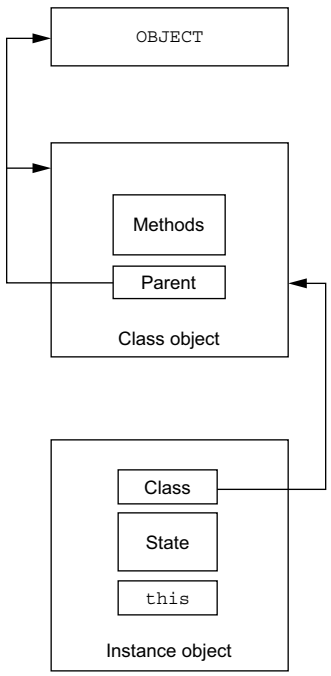

Figure 8.2 The minimal object system you've built implements a major portion of features that are supported by most common OO systems. Everything ultimately derives from a common entity called `OBJECT`. Instances look up the class they derive from and look up methods there. Methods can also be looked up in the chain of hierarchy.

Here's the call to the `:about` method that wasn't working a few paragraphs back:

```
(donna :about 3)
;=> "I was born about 21 years ago"
```

Again, remember to reevaluate everything after the last change, including the definitions of `Person`, `Woman`, and `donna` itself. Notice that the `:about` method is being called from the parent class `Person`. Further notice that the body of the `:about` method calls `:age` on the `this` reference, which is defined in both `Person` and `Woman`. Your object system correctly calls the one on `donna`, because you overrode the definition in the `Woman` class.

You've completed what you set out to do. You've written a simple object system that does most of what other languages provide. You can define classes, inherit from others, create instances, and call methods that follow the inheritance hierarchy. Objects can even refer to themselves using the `this` keyword. The following listing shows the complete code.

Listing 8.1 A simple object system for Clojure

```
(declare ^:dynamic this)
(declare find-method)
(defn new-object [klass]
  (let [state (ref {})]
    (fn thiz [command & args]
      (case command
        :class klass
        :class-name (klass :name)
        :set! (let [[k v] args]
                (dosync (alter state assoc k v))
                nil)
        :get (let [[key] args]
               (@state key))
        (let [method (klass :method command)]
          (if-not method
            (throw (RuntimeException.
                (str "Unable to respond to " command))))
          (binding [this thiz]
            (apply method args)))))))
(defn new-class [class-name parent methods]
  (fn klass [command & args]
    (case command
      :name (name class-name)
      :parent parent
      :new (new-object klass)
      :methods methods
      :method (let [[method-name] args]
                (find-method method-name klass)))))
(def OBJECT (new-class :OBJECT nil {}))
(defn find-method [method-name klass]
  (or ((klass :methods) method-name)
      (if-not (= #'OBJECT klass)
        (find-method method-name (klass :parent)))))
```

```
(defn method-spec [sexpr]
  (let [name (keyword (second sexpr))
    body (next sexpr)]
    [name (conj body 'fn)]))
(defn method-specs [sexprs]
  (->> sexprs
       (filter #(= 'method (first %)))
       (mapcat method-spec)
       (apply hash-map)))
(defn parent-class-spec [sexprs]
  (let [extends-spec (filter #(= 'extends (first %)) sexprs)
        extends (first extends-spec)]
    (if (empty? extends)
      'OBJECT
      (last extends))))
(defmacro defclass [class-name & specs]
  (let [parent-class (parent-class-spec specs)
        fns (or (method-specs specs) {})]
    `(def ~class-name (new-class '~class-name  #'~parent-class ~fns))))
```

So that's it, clocking in at a little over 50 lines of code. We haven't added several features that are provided by more robust object systems, but you can certainly do so. For instance, this function doesn't perform a lot of error checking. You should extend this code to add syntactic checks to see if the made-up syntax is being used correctly, for instance.

The key to this implementation is the lexical closure. It's up to you to take a stand on the old debate: Are objects a poor man's closures, or is it the other way around? More than anything else, though, it's important to remember this: although this example showed some of the power of functional programming (and that traditional OOP features aren't particularly special), in most cases, such artificial constructs are unnecessary in languages such as Clojure. We'll talk about why this is next.

DATA ABSTRACTION

We mentioned that constructs such as the object system aren't particularly useful in a language such as Clojure. There are two reasons for this.

The first relates to abstraction. Despite popular belief, you don't need objects to create data abstraction. There's a real alternative in Clojure in its core data structures: each implementation of the sequence abstraction (the hash map, the vector, and so on) is a suitable candidate to represent data in an application. Given that these are immutable and therefore thread safe, there's no strict need for procedural abstractions that wrap their mutation. Further, when you need to make a *new* abstraction of your own that existing types might want to participate in, you can use Clojure's protocol feature introduced in the next chapter.

The second reason is a bit more subjective. Alan Perlis once said that it's better to have 100 functions that operate on a single data structure instead of 10 functions that operate on 10 data structures.[1] Having a common data structure (in Clojure's case,

[1] "Epigrams on Programming," *SIGPLAN Notices* 17(9), September 1982. Archived at http://goo.gl/6PtaEm.

the sequence abstraction) allows for more code reuse, because code that works on sequences can be used no matter what specific data it contains. An example is the large sequence library, which works no matter what the specific implementation is.

Using the let-over-lambda technique shown previously has its value, as you saw in the earlier part of the chapter. But creating an object system like the one created in the second half raises a barrier of inoperability with other libraries that don't know about it. In conclusion, although the object system serves to illustrate the use of closures and some traditional OOP concepts, using the built-in core data structures is a much better choice in your own programs.

FINAL NOTES

It's worth recalling that the implementation of this little object system was done using functions. Fewer than half of the total lines of code were for manipulating the functions, and the rest were to make the syntax look a certain way. The macro `defclass` and the supporting functions (`parent-class-spec`, `method-specs`, and `method-spec`) account for over half the code. This syntactic layer sports an arbitrarily chosen syntax. The semantics of the object system wouldn't change if you decided to use a different syntax. The syntax isn't important (in this case certainly, and also in general); instead, it's the underlying mechanisms and features that make a piece of code useful. Having said that, a nice syntax certainly helps! The first reason is that it makes for convenient, intuitive use. The second is that you can write error checkers that analyze the code as data and give meaningful errors that are easier to understand and correct. You could certainly do more at the syntactic level to make this library easier to use.

Similarly, there are many features that you could add at a semantic level. For instance, there's no reason for a class hierarchy to be static. Instead of it being cast in stone when `defclass` is called, you could add messages that support modifying the hierarchy at runtime. As an example, your classes could respond to `:set-class!` and `:set-parent!`. Adding such features might be an interesting exercise for a spare afternoon.

8.4 Summary

This chapter was about functional programming, the understanding of which is crucial to programming in Clojure. If you're coming from an imperative background, this transition can take some effort. But the results are sweet, because functional code is more expressive, more reusable, and usually shorter.

We started out by having you create your own implementations of `map`, `reduce`, and `filter`—the workhorses of functional programming languages. Thinking recursively and in terms of lazy sequences is another important skill that will have a big impact on your Clojure programs.

We then looked at the technique of using partial function application to create specialized functions. To understand how partial application works, we explored lexical closures, another fundamental tool in the functional programmer's tool belt.

Once we presented a basic explanation of them, you gathered everything from this chapter (and from the ones so far!) to create your own little object system. This exercise was meant to demonstrate the power of closures and to also shed light on the fact that functional programming transcends traditional OOP.

Next we'll look at Clojure's protocols, which allow you to add function implementations specific to a particular object type.

PROGRAMMING

Clojure IN ACTION Second Edition

Rathore • Avila

Free eBook
SEE INSERT

Clojure is a modern Lisp for the JVM. It has the strengths you expect: first-class functions, macros, and Lisp's clean programming style. It supports functional programming, making it ideal for concurrent programming and for creating domain-specific languages. Clojure lets you solve harder problems, make faster changes, and end up with a smaller code base. It's no wonder that there are so many Clojure success stories.

Clojure in Action, Second Edition, is an expanded and improved version that's been updated to cover the new features of Clojure 1.6. The book gives you a rapid introduction to the Clojure language, moving from abstract theory to practical examples. You'll start by learning how to use Clojure as a general-purpose language. Next, you'll explore Clojure's efficient concurrency model, based on the database concept of Software Transactional Memory (STM). You'll gain a new level of productivity through Clojure DSLs that can run on the JVM. Along the way, you'll learn countless tips, tricks, and techniques for writing smaller, safer, and faster code.

What's Inside

- Functional programming basics
- Metaprogramming with Clojure's macros
- Interoperating with Java
- Covers Clojure 1.6

Assumes readers are familiar with a programming language like C, Java, Ruby, or Python.

Amit Rathore has 12 years of experience building large-scale, data-heavy applications for a variety of domains. **Francis Avila** is a software developer at Breeze with seven years of experience in back- and front-end web development.

"An excellent book for programmers of all levels of experience."
—Palak Mathur, Capital One

"A down-to-earth explanation of functional programming and one of the best introductory books on Clojure to date."
—Jeff Smith, Single Source Systems

"The right balance of idiomatic Clojure and how to use it pragmatically within the greater Java ecosystem."
—Scott M. Gardner, Multimedia LLC

"A quick, no-nonsense way to get up to speed with Clojure!"
—Jonathan Rioux, RGA Canada

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/clojure-in-action-second-edition

ISBN 13: 978-1-61729-152-4
ISBN 10: 1-61729-152-8
54999
9 781617 291524

MANNING $49.99 / Can $57.99 [INCLUDING eBOOK]