SAMPLE CHAPTER

Elegant applications on the JVM

Clojure IN ACTION

Amit Rathore

MANNING

Clojure in Action

by Amit Rathore

Chapter 13

brief contents

13 More on functional programming

This chapter covers

- A refresher on higher-order functions
- Partial application and currying of functions
- Lexical closures
- Traditional object-oriented programming in Clojure

So far, you've seen a lot of the Clojure programming language, and you've used it to write programs that can access data stores, communicate on message queues, consume and provide you services, crunch large amounts of data, and more. In this chapter, we'll revisit a fundamental concept of Clojure—that of functional programming.

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

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

Although partial application (and certainly currying) isn't particularly widespread in Clojure code, sometimes it's the perfect fit for the job.

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.

13.1 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 duplication in code. In this section, we'll look at a few simple examples of higher-order functions that can greatly simplify code. 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.

13.1.1 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:

```
(defn square [x]
  (* x x))

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

This works as expected, and you can test this at the REPL as follows:

```
user=> (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:

```
user=> (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 were applying a function to each element and were collecting the results before returning the list of return 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:

```
user> (do-to-all square [1 2 3 4 5 6])
(1 4 9 16 25 36)

user> (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. The `map` function is an abstraction that allows you to apply any function across sequences of arguments and collect results into another sequence. Our implementation suffers from a rather fatal flaw. The issue is that without tail-call optimization, it will blow the call stack if a long enough list of elements is passed in. Here's what it will look like:

```
user=> (do-to-all square (range 11000))
No message.
  [Thrown class java.lang.StackOverflowError]
```

This is because Clojure doesn't provide tail-call optimization (thanks to limitations of the JVM), but it does provide a way to fix this problem. 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))))))
```

Tail calls

A *tail call* is a call to a function from the last expression of another function body. When such a tail call returns, the calling function returns the value of the tail call. In most functional languages, tail calls are eliminated, so that using such calls doesn't consume the stack. This is possible because function calls from the tail position can be rewritten as jumps by the compiler or interpreter.

Clojure doesn't do this because of a related limitation on the JVM.

Now, because we made this return a lazy sequence, it no longer attempts to recursively compute all the elements to return. It now works as expected:

```
user=> (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 (the Clojure version does a lot more). The `map` function is an extremely useful higher-order function, and as you've seen over the last few chapters, it sees heavy usage. Let's now look at another important operation, which can be implemented using another higher-order function.

13.1.2 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 a total of a list of numbers or the largest number. Let's implement the total first:

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

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

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

Now let's write a function to return the greatest from a list of numbers. First, let's 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 we can use it to search for the largest number in a series of numbers:

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

Let's see if this works:

```
user> (greatest-of [5 7 9 3 4 1 2 8])
9
user> (greatest-of [])
nil
```

We have it working, but there's clearly some duplication in `total-of` and `greatest-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. Let's extract out the commonality into a function:

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

Now, we can easily use `compute-across` in order to implement `total-of` and `largest-of`:

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

(defn greatest-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:

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

user> (greatest-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, let's see if it works:

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

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

Our `compute-across` function is similar to something you've already seen: the `reduce` function that's part of Clojure's core functions. It allows you to process sequences of data and compute some final result. Let's now look at another related example of using our `compute-across`.

13.1.3 Filtering lists of things

We wrote a function in the previous section that allows us to collect all numbers greater than a particular threshold. Let's now 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 it is in action:

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

Notice how easy it is, now that we have our convenient little `compute-across` function (or the equivalent `reduce`). Also, notice that there's duplication in our `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. Let's 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:

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

To reimplement our 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. Our `select-if` function is another useful, low-level function that we can use with any sequence. In fact, Clojure comes with such a function, one that you've seen before: `filter`.

Over the last few pages, we'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 we did 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. Our `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.

13.2 Partial application and currying

We wrote several higher-order functions in the last section. Specifically, our functions accepted a function as one argument and applied it to other arguments. Now we're going to look at another kind of higher-order functions, those that create and return new functions. The ability to create functions dynamically is a crucial aspect of functional programming. In this section, we'll focus on a functional programming technique where we'll write functions that return new functions of less arity than the ones they accept as an argument. We'll do this by partially applying the function, the meaning of which will become clear shortly.

13.2.1 Adapting functions

Let's imagine you have a function that accepts a tax percentage (such as 8.0 or 9.75) and a price. It returns the price by adding the appropriate tax. You can easily implement this with 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:

```
user> (price-with-tax 9.5 100)
109.5
```

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 9.25 %) prices))
```

And you could then batch-calculate pricing with taxes:

```
user> (def prices [100 200 300 400 500])

user> (with-california-taxes prices)
(109.25 218.5 327.75 437.0 546.25)
```

Notice that in the definition of `with-california-taxes`, we created 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, we might have had to define a separate function that we may never have used anywhere else, such as this:

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

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

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

If we had to handle any more, the duplication would certainly get to us. 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 we have this meta function, we can remove the duplication you saw earlier, by defining state-specific functions as follows:

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

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

Notice again that we're creating new vars (and that we'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 we started out with a function of a certain arity (in this case, `price-with-tax` accepts two arguments), and we 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 OO literature). You don't need special ceremony to do this in Clojure, thanks to functions being first class. 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, in order to create a new function of k arguments. Let's create a function to illustrate this:

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

Now, in order 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))
```

Let's ensure that this function works as expected:

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

OK, 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. Here's a function that does this:

```
(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:

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

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

And you can see if these work as expected:

```
user> (of-2-args 4 5)
"abc45"

user> (of-3-args 3 4 5)
"ab345"
```

We called our new function `partially-applied` because it returns a function that's a partially applied version of the function we 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:

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

And here it is in action:

```
user> (of-2-args 4 5)
"abc45"

user> (of-3-args 3 4 5)
"ab345"
```

You now understand what it means to partially apply a function. 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 usage in the next section.

13.2.2 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 we pass the `compute-across` function an empty vector as the last argument. We'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 our new function as follows:

```
user> (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:

```
user> (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. Let's now further abstract this idea of filtering results up or down as follows:

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

Here, we've created a new function using `partial`. We 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 ()))
```

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

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

user> (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. In the next section, we'll discuss a related functional programming concept called currying.

13.2.3 Currying

Currying is the process by which a function that takes multiple arguments is transformed into another that takes a single argument and returns a partially applied function. As you can imagine, the concepts of currying and of partial application are closely related. To understand this better, consider the following function:

```
(defn add-pair [a b]
  (+ a b))
```

Imagine that the `add-pair` function was curried. If you were to call it with only a single argument, you'd get back a new function of one argument, for example:

```
(def inc-by-two (add-pair 2))
;; this won't actually work, since Clojure functions are not curried
```

This new function would add 2 to any argument it was passed, for instance:

```
user> (inc-by-two 3)
5
```

Curried functions are particularly central to certain functional programming languages such as Haskell that have functions of only one argument. In such languages, a function such as `add-pair` would work the following way:

```
user> (add-pair 2 3)
=> ((anonymous-function-that-adds-two) 3)
=> 5
```

The programmer wouldn't have to do anything explicit in order to invoke such behavior; all functions are curried by default in such languages. Okay, back to Clojure. Trying the previous example on the REPL will fail because `add-pair` isn't curried and it needs two arguments when called. You could accomplish what we're describing here using `partial`, but the semantics would be different:

```
user> (def inc-by-two (partial add-pair 2))
#'user/inc-by-two
```

This would then work as expected:

```
user> (inc-by-two 3)
5
```

This difference (having to be explicit about partially applying arguments) is because Clojure functions aren't automatically curried. You might be able to write a function as follows in order to remedy this situation:

```
(defn curry-1 [f]
  (fn [x]
    (fn [y]
      (f x y))))
```

Now you could try our previous example again by first creating a curried version of `add-pair` as shown here:

```
user> (def add-pair-curried (curry-1 add-pair))
#'user/add-pair-curried
```

This creates a curried version of `add-pair` that will work as desired when applied to a single argument:

```
user> (def inc-by-two (add-pair-curried 2))
#'user/inc-by-two
```

Now `inc-by-two` will also work as you saw before:

```
user> (inc-by-two 3)
5
```

`curry-1` worked for us because `add-pair` takes two arguments. What if you had more arguments, such as the following function?

```
(defn add-quadruple [a b c d]
  (+ a b c d))
```

`curry-1` won't suffice to convert this into a curried function. Because Clojure functions can accept a variable number of arguments, you can write a slightly more generalized version of `curry`. Strictly speaking, currying returns single-argument functions because the concept applies more to languages that have only single-argument functions. Still, our function here is more applicable to languages such as Clojure that do support more than one argument. Here it is:

```
(defn curry-2 [f]
  (fn [& args]
    (fn [& more-args]
      (apply f (concat args more-args)))))
```

With this version of `curry`, you can create curried versions of functions that take more than two arguments as well, for instance:

```
user> (def add-quadruple-curried (curry-2 add-quadruple))
#'user/add-quadruple-curried
```

Now, you can use our curried version of the function to get new functions when applied to fewer than four arguments:

```
user> (def add-2-to-triple (add-quadruple-curried 2))
#'user/add-2-to-triple
```

And here's how you can use it:

```
user> (add-2-to-triple 3 4 5)
14
```

`add-quadruple-curried` will also work when passed more than one argument, as shown here:

```
user> (def add-5-to-pair (add-quadruple-curried 2 3))
#'user/add-5-to-pair

user> (add-5-to-pair 4 5)
14
```

So far, we haven't done anything too major that couldn't be accomplished with `partial`. If all functions were automatically curried, you'd be able to do something like the following:

```
(def inc-by-9 (add-5-to-pair 4))
;; won't work since add-5-to-pair isn't curried
```

So even though we created `add-5-to-pair` by calling a curried function with less than complete arguments, the returned function itself isn't curried. `add-5-to-pair` behaves like all regular Clojure functions in that it needs to be called with all its arguments (or passed to `partial` explicitly).

Let's now try to write a version of `curry` that returns curried functions if required. We'll have to keep track of how many parameters such a function takes so we can

decide whether all arguments have been passed. The function `curried-fn` shown here does this:

```
(defn curried-fn [func args-len]
  (fn [& args]
    (let [remaining (- args-len (count args))]
      (if (zero? remaining)
        (apply func args)
        (curried-fn (apply partial func args) remaining)))))
```

Before using this function, we'll also create a wrapper around it so that we have a syntactically clear way to create curried functions. The following macro, `defcurried`, does the job:

```
(defmacro defcurried [fname args & body]
  `(let [fun# (fn ~args (do ~@body))]
     (def ~fname (curried-fn fun# ~(count args)))))
```

Now we're ready to try this out. Let's try our `add-quadruple` function again:

```
(defcurried add-quadruple [a b c d]
  (+ a b c d))
```

Now we have a curried function that returns either the result of the computation as specified in its body or another function that's the result of partially applying the supplied arguments but also curried. Here's an example:

```
user> (def add-5-to-triple (add-quadruple 5))
#'user/add-5-to-triple
```

You've seen this before, when we used the `add-quadruple-curried` version of this function. The difference that time was that the resulting `add-5-to-triple` wasn't curried but was a regular Clojure function. This time around, we can do the following:

```
user> (def add-9-to-pair (add-5-to-triple 4))
#'user/add-9-to-pair
```

And it doesn't end there. We can go further:

```
user> (def inc-by-12 (add-9-to-pair 3))
#'user/inc-by-12
```

And finally, we can use the `inc-by-12` function:

```
user> (inc-by-12 3)
15
```

So, we've added the ability to automatically curry our functions. Figure 13.1 shows the difference between Clojure's `partial` function and our new `curry` function.

This functionality is limited, to be sure. For instance, it can't handle a variable number of arguments. Even so, why would we need such a feature? Clojure has two things that reduce the necessity of such automatic currying: `partial` and anonymous functions. Between these two and the fact that Clojure functions aren't restricted to being of a single arity, automatic currying is less of a requirement. Figure 13.1 shows

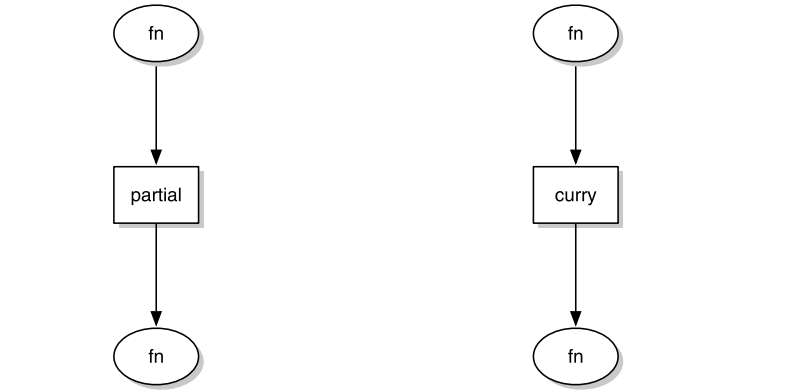

Figure 13.1 Although both `partial` and `curry` return functions, a partially applied function is always ready to run when it's applied to the remaining arguments. If given insufficient arguments, a partially applied function will throw an exception complaining about that. Curried functions return functions that are further curried if they're applied to insufficient arguments.

the difference between partial and currying in a pictorial way. Our next stop will be to look at using anonymous functions to similar effect.

ANONYMOUS FUNCTIONS

The reader macro for anonymous functions is convenient shorthand for creating functions that are partially applied. Let's recall our regular function `add-pair`, whose job is to add the two numbers passed to it. Consider the following:

```
(def inc-by-3 #(add-pair 3 %1))
```

You know what this does; `inc-by-3` is now a single-argument function that will add 3 to its argument. Note that this is equivalent to the following use of the `partial` function:

```
(def inc-by-3 (partial add-pair 3))
```

It's also similar to the use of the curried `add-pair-curried`:

```
(def inc-by-3 (add-pair-curried 3)
```

The anonymous function requires a little more typing, but it's more expressive. Not only can it handle variable args (via `%&`), but it can also handle arguments in specific places, for instance:

```
user> (def half-of #(/ %1 2))
#'user/half-of

user> (half-of 10)
5
```

You couldn't use partial (or curried functions) to do this, as shown here

```
user> ((partial / 2) 10)
1/5
```

because `partial` associates arguments in order (left to right). The same is true with curried functions. The anonymous function reader macro, on the other hand, is explicit about the order in which functions are applied. We mentioned that you could handle functions with variable arguments; let's look at an example. Consider the following function of variable arity:

```
(defn add-numbers [& numbers]
  (apply + numbers))
```

Now, you could create partially applied versions of this function using anonymous functions:

```
user> (def add-5-to-remaining #(apply add-numbers 2 3 %&))
#'user/add-5-to-remaining
```

Here it is in action:

```
user> (add-5-to-remaining 1 4 5)
15
```

As noted earlier, you couldn't have done this with our `defcurried` macro or using any related technique. A final point for Clojure's anonymous functions is that the reader macro syntax makes it abundantly clear that a function object is being constructed, whereas it isn't as clear with curried functions. Having said all that, there may well be cases where having the behavior of curried functions is what you need.

You've already seen closures, in previous chapters of this book as well as in this section. In the next section, we're going to explore them in more detail.

13.3 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 object-oriented sense of the word). At the end, we'll create a little object-oriented layer for Clojure in order to demonstrate that Clojure (and Lisps in general) go beyond traditional ideas of object-oriented programming.

13.3.1 Free variables and closures

Before we jump into what closures are, let's look at the concept of free variables. A free variable is a placeholder for a value that will be substituted in at some later point. More specifically, given a lexically scoped block of code, a free variable is one that's neither an argument nor a local variable, for instance:

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

Here, `num1` and `num2` are arguments of the `adder` function and so 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` is not a free variable. Finally, an anonymous function is created that accepts an argument called `y`. Within this anonymous function, `y` is not a free variable, but `x` is.

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

```
user> (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:

```
user> (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. Now let's look at what closures can do for our Clojure programs.

13.3.2 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))
```

You know what this code will do: a nice divide-by-zero exception will be thrown. 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)))))
```

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 said that Clojure evaluates function arguments in an eager manner. This behavior is an advantage of macros in that macros don't evaluate arguments, allowing us as programmers to be in control. But the function `try-catch` here, achieves the same intended effect by accepting functions that will be evaluated later. This is the reason we created anonymous functions that we then passed in as arguments. Because the function enclosed free variables `x` and `y`, 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 extremely useful aspect of closures.

13.3.3 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 the concept of users in your application and that you wanted to store their login information, along with their email address. Consider the following function:

```
(defn new-user [login password email]
  (fn [a]
    (condp = a
      :login login
      :password password
      :email email)))
```

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

```
user> (def anya (new-user "anya" "secret" "anya@currylogic.com"))
#'user/anya

user> (anya :login)
"anya"

user> (anya :password)
"secret"

user> (anya :email)
"anya@currylogic.com"
```

First, we created a new function object by calling the `new-user` function with `"anya"`, `"secret"`, and `"anya@currylogic.com"` as arguments. Then, we were able to query it for the login, password, and email address. `anya` 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 `anya`. This is a form of message passing: the keywords are messages that we're sending to a receiver object, which in this case is the function object `anya`. 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]
    (condp = a
      :login login
      :email email)))
```

With this constructor of new users, there's no way to directly access the password a user was created with. Having come this far, we'll take a short break to compare functions such as `anya` 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 `anya` a function or is it a kind of data object? It certainly behaves similarly to a hash map, where one can query the value associated with a particular key. In this case, we wrote code to expose only those keys that we considered public information, while hiding the private pieces. Because `anya` 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 `anya` 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 our user objects. We'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]
    (condp = a
      :login login
```

```
      :email email
      :authenticate (= password (first args)))))
```

Now let's try it out:

```
user> (def adi (new-user "adi" "secret" "adi@currylogic.com"))
#'user/adi

user> (adi :authenticate "blah")
false

user> (adi :authenticate "secret")
true
```

Our 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 object-oriented programming 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 usage 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.

13.3.4 An object system for Clojure

In the previous section, we 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. We 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 object-oriented languages call classes. We'll allow a simple class hierarchy, and we'll let objects refer to themselves by providing a special symbol, traditionally named, `this`. Finally, we'll wrap this functionality in a syntactic skin in order to make it appear more familiar.

DEFINING CLASSES

We'll start simple. We'll create the ability to define classes that have no state or behavior. We'll lay the foundations of our object system starting with the ability to define a class that's empty. For instance, we'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. We 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 we're defining. Once we implement this, we can start adding more functionality to our little object system. Consider the following implementation:

```
(defn new-class [class-name]
  (fn [command & args]
    (condp = command
      :name (name class-name))))

(defmacro defclass [class-name]
  `(def ~class-name (new-class '~class-name)))
```

So our little `Person` class is a function that closes over the `class-name`, passed in along with a call to `defclass`. Our classes support a single message right now, which is `:name`. When passed `:name`, our `Person` function returns the string `"Person"`. This allows us to do the following:

```
user> (def some-class Person)
#'user/some-class

user> (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 our classes are closures, our instantiated classes will also be closures. We'll create this with a function called `new-object` that will accept the class that we'd like to instantiate. Here's the implementation:

```
(defn new-object [klass]
  (fn [command & args]
    (condp = 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

```
user> (cindy :class)
#<user$new_class__4224$fn__4226 user$new_class__4224$fn__4226@26ced1a8>
```

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

```
user> ((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]
    (condp = command
      :class klass
      :class-name (klass :name))))
```

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

```
user> (def cindy (new-object Person))
#'user/cindy

user> (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, we've already set up our class as a function that can handle incoming messages, so we'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]
    (condp = command
      :name (name class-name)
      :new (new-object klass))))
```

Notice that `new-object` accepts a class, and we need to refer to the class object from within it. We 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:

```
user> (def nancy (Person :new))
#'user/nancy

user> (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 our objects to maintain state.

OBJECTS AND STATE

In chapter 6, we explored Clojure's support for managing state. In this section, we'll use one of those available mechanisms to allow objects in our object system to also become stateful. We'll use a ref so that our objects will be able to participate in coordinated transactions. We'll also expand the vocabulary of messages that our objects can understand by supporting `:set!` and `:get`. They'll allow us to set and fetch values from our objects, respectively.

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

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

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

```
user> (def nancy (Person :new))
#'user/nancy

user> (nancy :set! :name "Nancy Warhol")
nil

user> (nancy :get :name)
"Nancy Warhol"
```

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

```
user> (nancy :set! :name "Nancy Drew")
nil

user> (nancy :get :name)
"Nancy Drew"
```

We're coming along in our 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. The syntax we'd like to support is to include the methods along with the class definition so as to keep it together. This is another common pattern with languages such as Java and C++. The following is the syntax we'd like to be able to use:

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

We'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 we've laid out the expected syntax, we can go about the implementation. First, we'll work on `defclass`, in order 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:

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

Because you're going to specify more than one method inside your 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. Here it is:

```
(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:

```
user> (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))}
```

We 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 our `new-class` function for later use. Here's the associated revision of `new-class`:

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

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

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

Now, our desired syntax from before will work:

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

So we've successfully made it possible to specify methods along with our class definitions. Now, all you have to do is to extend our objects so that you can invoke these methods. We'll do this in the next section.

INVOKING METHODS

In order to be able to invoke a method on one of our 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 our classes, which would accept the name of the method you're looking for. Consider the following revision to our `new-class` function:

```
(defn new-class [class-name methods]
  (fn klass [command & args]
    (condp = 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. In order to find a method from our 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 our all other messages. Here's an example:

```
(Person :method :age)
#<user$age__3921 user$age__3921@1b4865b1>
```

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 we laid out in the class definition:

```
user> ((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 we've built so far in order 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]
     (condp = command
       :class klass
       :class-name (klass :name)
       :set! (let [[k v] args]
               (dosync (alter state assoc k v))
               nil)
       :get (let [[key] args]
              (key @state))
       (let [method (klass :method command)]
         (if-not method
           (throw (RuntimeException.
                   (str "Unable to respond to " command))))
         (apply method args))))))
```

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

```
user> (def shelly (Person :new))
#'user/shelly

user> (shelly :age)
20

user> (shelly :greet "Nancy")
"Hello there, Nancy"
```

Remember, in order 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. Our next little feature is the ability of our objects to refer to themselves.

Referring to this

So far, our 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. We'll support the same functionality by providing a special name, which we'll also call `this`.

Once we're finished, we'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. In order to implement this, we'll first create a var named `this`, so that our class definitions continue to work (without complaining about unresolved symbols).

```
(declare 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 we have something to bind to. We will employ the same trick we did when we 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]
      (condp = command
        :class klass
        :class-name (klass :name)
        :set! (let [[k v] args]
                (dosync (alter state assoc k v))
                nil)
        :get (let [[key] args]
               (key @state))
        (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. You can confirm that it works on the REPL:

```
user>  (def shelly (Person :new))
#'user/shelly

user> (shelly :about 2)
"I was born about 22 years ago"
```

Remember to evaluate the new, revised definition of `new-object` before trying it out. We've added almost all the features we wanted to when we started this section. The last thing we'll add is the ability of a class to inherit from another.

CLASS INHERITANCE

We're about to add a final feature to our little object system. Traditional OO programming 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, we'll add the ability to do that.

The first thing we'll do is allow our class definition to specify the parent class. Imagine that our syntax will look like this:

```
(defclass Woman
    (extends Person)
```

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

Here, our new `Woman` class inherits from a previously defined `Person` class. We've used the term `extends` to signify this relationship, as is common to other OO languages. Okay, now that we have our notation, let's implement it. The first step is to write a function that can extract the parent class information from the class definition. Before we can do so, we have to decide what to do if one isn't provided.

Again, we'll look at other languages for the answer. Our 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, we'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:

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

Now that we have the parent class, we'll pass it to our `new-class` function. We 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` of that name by using the `var` special form:

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

```
(defn new-class [class-name parent methods]
  (fn klass [command & args]
    (condp = 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 we can use our new `defclass`. We're looking up the parent class using `var`, so we'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. Let's check this on the REPL:

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

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

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

We've only half finished the job we started out to do. Although we can specify the parent class in our calls to `defclass`, method calls on our objects won't work right with respect to parent classes. The following illustrates the problem:

```
user>(def donna (Woman :new))
#'user/donna

user>(donna :greet "Shelly")
"Hello, Shelly"

user>(donna :age)
18

user>(donna :about 3)
; Evaluation aborted.
Unable to respond to :about
  [Thrown class java.lang.RuntimeException]
```

In order to fix this last error, we'll have to improve our method lookup in order to find the method in the parent class. Indeed, we'll have to search up the hierarchy of classes (parent of the parent of the parent…) until we hit `OBJECT`. We'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)))))
```

In order for this to work, we'll need to have our classes handle another message, namely, `:methods`. Also, our classes will use this new version of `find-method` in order to perform method lookup. Here's the updated code:

```
(defn new-class [class-name parent methods]
  (fn klass [command & args]
    (condp = 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, our object system will work as planned. Figure 13.2 shows the conceptual model of the class system we've built.

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

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

Again, remember to reevaluate everything after the last change, including the definition of `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

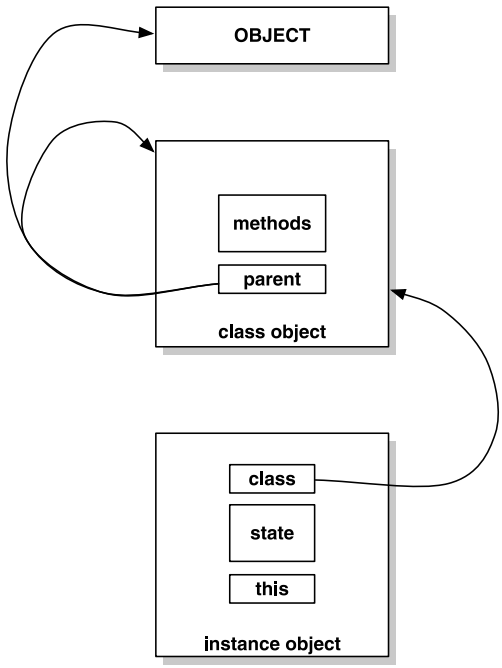

Figure 13.2 The minimal object system we've built implements a major portion of features that are supported by most common OO systems. Everything ultimately derives from a common entity called OBJECT. Instances look up the class they derive from and look up methods there. Methods can also be looked up in the chain of hierarchy.

`this` reference, which is defined in both `Person` and `Woman`. Our object system correctly calls the one on `donna`, because we overrode the definition in the `Woman` class.

We've completed what we set out to do. We've written a simple object system that does most of what other languages provide. We 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 13.1 A simple object system for Clojure

```
(declare this find-method)

(defn new-object [klass]
  (let [state (ref {})]
    (fn thiz [command & args]
      (condp = command
        :class klass
        :class-name (klass :name)
        :set! (let [[k v] args]
                (dosync (alter state assoc k v))
                nil)
        :get (let [[key] args]
               (key @state))
        (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]
    (condp = 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. The key to this implementation is the lexical closure. It's left 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 little example showed some of the power of functional programming (and that traditional object-oriented programming features aren't particularly special), in most cases, such artificial constructs are unnecessary in languages such as Clojure. We'll talk about this briefly next.

Data abstraction

We said that constructs such as our 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.

The second reason is a bit more subjective. Alan Perlis once said that it's better to have a 100 functions that operate on a single data structure instead of 10 functions that operate on 10 data structures. Having a common data structure (in Clojure's case, 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 we 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 usage 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 we 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 usage. The second is that you can write error checkers by analyzing 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, our classes could respond to `:set-class!` and `:set-parent!`. Adding such features might be an interesting exercise for a spare afternoon.

13.4 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 creating our 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 partial application of functions and the associated concept of currying. Although these aren't used that often in Clojure, the insight you gleaned from these concepts can help when they do fit.

Our final stop was to explore closures, another fundamental tool in the functional programmer's tool belt. Once we presented a basic explanation of them, we gathered everything from this chapter (and from the ones so far!) to create our 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 object-oriented programming.

PROGRAMMING

Clojure IN ACTION

Amit Rathore

Clojure is a modern Lisp for the JVM, and it has the strengths you'd expect: first-class functions, macros, support for functional programming, and a Lisp-like, clean programming style. But it's not enough to learn a language, you also need to know what to do with it.

Clojure in Action is a practical guide focused on applying Clojure to practical programming challenges. You'll start with a language tutorial written for readers who already know OOP. Then, you'll dive into the use cases where Clojure really shines: state management, safe concurrency and multicore programming, first-class code generation, and Java interop. In each chapter, you'll first explore the unique characteristics of a problem area and then discover how to tackle them using Clojure. Along the way, you'll explore practical matters like architecture, unit testing, and setup as you build a scalable web application that includes custom DSLs, Hadoop, HBase, and RabbitMQ.

What's Inside

- A fast-paced Clojure tutorial
- Creating web services with Clojure
- Scaling through messaging
- Creating DSLs with Clojure's macro system
- Test-driven development with Clojure
- Distributed programming with Clojure, and more

This book assumes you're familiar with an OO language like Java, C#, or C++ but requires no background in Lisp or Clojure itself.

Amit Rathore is a VP of Engineering with a decade of experience building highly performant data-heavy web applications.

For access to the book's forum and a free eBook for owners of this book, go to manning.com/ClojureinAction

Free eBook
SEE INSERT

"An easy to read book and a great way to get up-to-speed on Clojure."
—Craig Smith, Suncorp

"Down-to-earth and thorough."
—Stuart Caborn, BNP Paribas

"Explains functional programming with Java."
—Doug Warren
Java Web Services

"An intriguing mix of emerging tech."
—Andrew Oswald
Chariot Solutions

"Teaches you how to put Clojure into action at warp speed!"
—Baishampayan Ghose (BG)
Qotd, Inc.

ISBN 13: 978-1-935182-59-7
ISBN 10: 1-935182-59-5
54999
9 781935 182597

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]