1 Have a Beginner's Mind

WHAT'S IN THIS CHAPTER?

- Understanding the differences between imperative and functional programming
- Learning how to think more functionally
- Discovering Clojure's unique perspective on object-oriented programming

If your mind is empty, it is always ready for anything, it is open to everything. In the beginner's mind there are many possibilities, but in the expert's mind there are few.

—Shunryu Suzuki

Over the past thirty years many popular programming languages have more in common with each other than they have differences. In fact, you could argue that once you have learned one language, it's not difficult to learn another. You merely have to master the subtle differences in syntax, and maybe understand a new feature that isn't present in the language that you're familiar with. It's not difficult to call yourself a polyglot programmer when many of the top languages in use today are all so similar.

Clojure, on the other hand, comes from a completely different lineage than most of the popular languages in use today. Clojure belongs to the Lisp family of programming languages, which has a very different syntax and programming style than the C-based languages you are probably familiar with. You must leave all of your programming preconceptions behind in order to gain the most from learning Clojure, or any Lisp language in general.

Forget everything you know, or think you know about programming, and instead approach it as if you were learning your very first programming language. Otherwise, you'll just be learning a new syntax, and your Clojure code will look more like Java/C/Ruby and less like Clojure is designed to look. Learning Clojure/Lisp will even affect the way you write in other languages, especially with Java 8 and Scala becoming more popular.

FUNCTIONAL THINKING

C, C++, C#, Java, Python, Ruby, and even to some extent Perl, all have very similar syntax. They make use of the same programming constructs and have an emphasis on an imperative style of programming. This is a style of programming well suited to the von Neumann architecture of computing that they were designed to execute in. This is probably most apparent in the C language, where you are responsible for allocating and de-allocating memory for variables, and dealing directly with pointers to memory locations. Other imperative languages attempt to hide this complexity with varying degrees of success.

> *In computer science, **imperative programming** is a **programming** paradigm that uses statements that change a program's state.*

This C-style of programming has dominated the programming scene for a very long time, because it fits well within the dominant hardware architectural paradigm. Programs are able to execute very efficiently, and also make efficient use of memory, which up until recently had been a very real constraint. This efficiency comes at the cost of having more complex semantics and syntax, and it is increasingly more difficult to reason about the execution, because it is so dependent upon the state of the memory at the time of execution. This makes doing concurrency incredibly difficult and error prone. In these days of cheap memory and an ever growing number of multiple core architectures, it is starting to show its age.

Functional programming, however, is based on mathematical concepts, rather than any given computing architecture. Clojure, in the spirit of Lisp, calls itself a general-purpose language; however, it does provide a number of functional features and supports the functional style of programming very well. Clojure as a language not only offers simpler semantics than its imperative predecessors, but it also has arguably a much simpler syntax. If you are not familiar with Lisp, reading and understanding Clojure code is going to take some practice. Because of its heavy focus on immutability, it makes concurrency simple and much less error prone than having to manually manage locks on memory and having to worry about multiple threads reading values simultaneously. Not only does Clojure provide all of these functional features, but it also performs object-oriented programming better than its Java counterpart.

Value Oriented

Clojure promotes a style of programming commonly called "value-oriented programming." Clojure's creator, Rich Hickey, isn't the first person to use that phrase to describe functional programming, but he does an excellent job explaining it in a talk titled *The Value of Values* that he gave at Jax Conf in 2012 (`https://www.youtube.com/watch?v=-6BsiVyC1kM`).

By promoting this style of value-oriented programming, we are focused more on the values than mutable objects, which are merely abstractions of places in memory and their current state. Mutation belongs in comic books, and has no place in programming. This is extremely powerful, because it allows you to not have to concern yourself with worrying about who is accessing your data and when. Since you are not worried about what code is accessing your data, concurrency now becomes much more trivial than it ever was in any of the imperative languages.

One common practice when programming in an imperative language is to defensively make a copy of any object passed into a method to ensure that the data does not get altered while trying to use it. Another side effect of focusing on values and immutability is that this practice is no longer necessary. Imagine the amount of code you will no longer have to maintain because you'll be using Clojure.

In object-oriented programming, we are largely concerned with information hiding or restricting access to an object's data through encapsulation. Clojure removes the need for encapsulation because of its focus on dealing with values instead of mutable objects. The data becomes semantically transparent, removing the need for strict control over data. This level of transparency allows you to reason about the code, because you can now simplify complex functions using the substitution model for procedure application as shown in the following canonical example. Here we simplify a function called `sum-of-squares` through substituting the values:

```
(defn square [a] (* a a))
(defn sum-of-squares [a b] (+ (square a) (square b))

; evaluate the expression (sum-of-squares 4 5)

(sum-of-squares 4 5)
(+ (square 4) (square 5))
(+ (* 4 4) (* 5 5))
(+ 16 25)
41
```

By favoring functions that are referentially transparent, you can take advantage of a feature called memorization. You can tell Clojure to cache the value of some potentially expensive computation, resulting in faster execution. To illustrate this, we'll use the Fibonacci sequence, adapted for Clojure, as an example taken from the classic MIT text *Structure and Interpretation of Computer Programs (SICP).*

```
(defn fib [n]
  (cond
    (= n 0) 0
    (= n 1) 1
    :else (+ (fib (- n 1))
             (fib (- n 2)))))
```

If you look at the tree of execution and evaluate the function for the value of 5, you can see that in order to calculate the fifth Fibonacci number, you need to call `(fib 4)` and `(fib 3)`. Then, to calculate `(fib 4)`, you need to call `(fib 3)` and `(fib 2)`. That's quite a bit of recalculating values that you already know the answer to (see Figure 1-1).

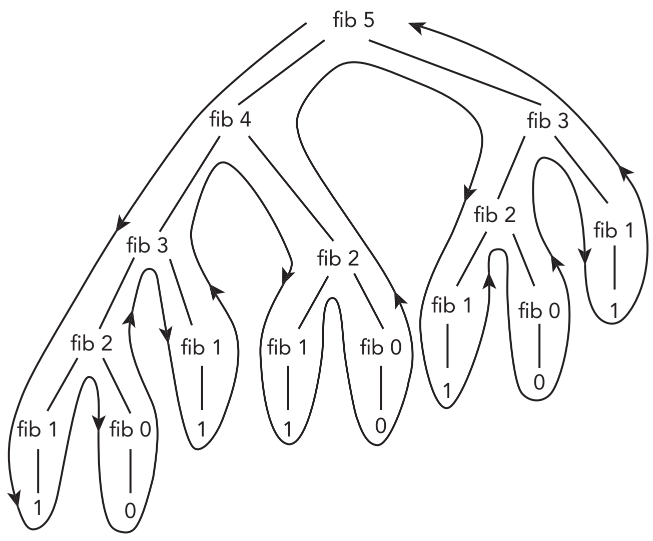

FIGURE 1-1

Calculating for `(fib 5)` executes quickly, but when you try to calculate for `(fib 42)` you can see that it takes considerably longer.

```
(time (fib 42))
"Elapsed time: 11184.49583 msecs"
267914296
```

You can rewrite a function to leverage memorization to see a significant improvement in the execution time. The updated code is shown here:

```
(def memoized-fib
  (memoize (fn [n]
             (cond
               (= n 0) 0
               (= n 1) 1
               :else (+ (fib (- n 1))
                        (fib (- n 2)))))))
```

When you first run this function, you'll see how the execution doesn't happen any faster; however, each subsequent execution instantaneously happens.

```
user> (time (memoized-fib 42))
"Elapsed time: 10586.656667 msecs"
```

```
267914296
user> (time (memoized-fib 42))
"Elapsed time: 0.10272 msecs"
267914296
user> (time (memoized-fib 42))
"Elapsed time: 0.066446 msecs"
267914296
```

This is a risky enhancement if you do this with a function that relies on a mutable shared state. However, since our functions are focused on values, and are referentially transparent, you can leverage some cool features provided by the Clojure language.

Thinking Recursively

Recursion is not something that is taught much in most imperative languages. Contrast this with most functional languages, and how they embrace recursion, and you will think more recursively. If you are unfamiliar with recursion, or struggle to understand how to think recursively, you should read *The Little Schemer*, by Daniel P. Friedman and Matthias Felleisen. It walks you through how to write recursive functions using a Socratic style of teaching, where the two authors are engaged in a conversation and you get to listen in and learn.

Let's take a look at a trivial example of calculating a factorial. A typical example in Java might look like the code shown below. You start by creating a local variable to store the ultimate result. Then, loop over every number, one-by-one, until you reach the target number, multiplying the last result by the counter variable defined by the `for` loop, and mutating the local variable.

```
public long factorial(int n) {
  long product = 1;
  for ( int i = 1; i <= n; i++ ) {
    product *= i;
  }
  return product;
}
```

Because of Clojure's focus on values and immutable structures, it relies on recursion for looping and iteration. A naïve recursive definition of a factorial in Clojure may look like the following:

```
(defn factorial [n]
     (if (= n 1)
          1
          (* n (factorial (- n 1)))))
```

If you trace the execution of the factorial program with an input of 6, as shown next, you see that the JVM needs to maintain each successive operation on the stack as `n` increases, until the factorial reaches a point where it returns a value instead of recurring. If you're not careful, you'll likely end up with a stack overflow. This style of recursion is often called linear recursion.

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

To resolve this dilemma, rewrite your function in a tail recursive style using the special operator called `recur,` which according to the documentation, "does constant-space recursive looping by rebinding and jumping to the nearest enclosing loop or function frame." This means that it tries to simulate a tail call optimization, which the JVM doesn't support. If you rewrite the preceding factorial example using this style, it looks something like the following:

```
(defn factorial2 [n]
    (loop [count n acc 1]
       (if (zero? count)
            acc
          (recur (dec count) (* acc count)))))
```

In this version of the factorial function, you can define an anonymous lambda expression using the `loop` construct, thus providing the initial bindings for the local variables `count` to be the value passed into the factorial function to start the accumulator at 1. The rest of the function merely consists of a conditional that checks the base case and returns the current accumulator, or makes a recursive call using `recur`. Notice how in the tail position of the function this program doesn't require the runtime to keep track of any previous state, and can simply call `recur` with the calculated values. You can trace the execution of this improved version of the factorial as seen here:

```
(factorial2 6)
(loop 6 1)
(loop 5 6)
(loop 4 30)
(loop 3 120)
(loop 2 360)
(loop 1 720)
720
```

The call to `factorial2` take fewer instructions to finish, but it doesn't need to place new calls on the stack for each iteration, like the first version of `factorial` did.

But what happens if you need to perform mutual recursion? Perhaps you want to create your own version of the functions for determining if a number is odd or even. You could define them in terms of each other. A number is defined as being even if the decrement of itself is considered odd. This will recursively call itself until it reaches the magic number of 0, so at that point if the number is even it will return true. If it's odd it will return false. The code for the mutually recursive functions for `my-odd?` and `my-even?` are defined here:

```
(declare my-odd? my-even?)

(defn my-odd? [n]
  (if (= n 0)
    false
    (my-even? (dec n))))

(defn my-even? [n]
  (if (= n 0)
    true
    (my-odd? (dec n))))
```

This example suffers from the same issue found in the first example, in that each successive recursive call needs to store some sort of state on the stack in order to perform the calculation, resulting in a stack overflow for large values. The way you avoid this problem is to use another special operator called trampoline, and modify the original code to return functions wrapping the calls to your recursive functions, like the following example:

```
(declare my-odd? my-even?)

(defn my-odd? [n]
  (if (= n 0)
    false
    #(my-even? (dec n))))

(defn my-even? [n]
  (if (= n 0)
    true
    *(my-odd? (dec n))))
```

Notice the `declare` function on the first line. We can call the function using the trampoline operator as shown here:

```
(trampoline my-even? 42)
```

If the call to a function, in this case `my-even?`, would return another function, `trampoline` will continue to call the returned functions until an atomic value gets returned. This allows you to make mutually recursive calls to functions, and not worry about blowing the stack. However, we're still left with one problem. If someone wishes to use the version of `my-even?` and `my-odd?`, they must have prior knowledge to know they must call them using `trampoline`. To fix that you can rewrite the functions:

```
(defn my-even? [n]
  (letfn [(e? [n]
            (if (= n 0)
              true
              #(o? (dec n))))
          (o? [n]
            (if (= n 0)
              false
              #(e? (dec n))))]
    (trampoline e? n)))

(defn my-odd? [n]
  (not (my-even? n)))
```

We've effectively hidden away the knowledge of having to use `trampoline` from our users.

Higher Order Functions

One of the many qualities that define a language as being functional is the ability to treat functions as first class objects. That means functions can not only take values as parameters and return values, but they can also take functions as parameters as well. Clojure comes with a number of commonly used higher order functions such as `map`, `filter`, `reduce`, `remove` and `iterate`, as well as providing you with the tools to create your own.

In Java, for example, if you want to filter a list of customers that live in a specific state, you need to create a variable to hold the list of filtered customers, manually iterate through the list of customers, and manually add them to the local variable you created earlier. You have to specify not only what you want to filter by, but also how to iterate through the list.

```
public List<Customer> filterByState(List<Customer> input, String state) {
     List<Customer> filteredCustomers = new ArrayList<>();

     for(Customer customer : input) {
          if (customer.getState().equals(state)) {
               filteredCustomers.put(customer);
          }
     }

     return filteredCustomers;
}
```

This Clojure example deals less with how to do the filtering, and is a bit more concise and declarative. The syntax may look a little strange, but you are simply calling the filter function with an anonymous function telling what you should filter on and finally the sequence you want to filter with.

```
(def customers [{:state "CA" :name "Todd"}
                {:state "MI" :name "Jeremy"}
                {:state "CA" :name "Lisa"}
                {:state "NC" :name "Rich"}])
(filter #(= "CA" (:state %)) customers)
```

One common design pattern that exists in object-oriented programming, the Command pattern, exists as a way to cope with the lack of first class functions and higher order functions. To implement the pattern, you first define an interface that defines a single method for executing the command, a sort of pseudo-functional object. Then you can pass this Command object to a method to be called at the appropriate time. The downfall of this is that you need to either define several concrete implementations to cover every possible piece of functionality you would need to execute, or define an anonymous inner class wrapping the functionality.

```
public void wrapInTransaction(Command c) throws Exception {
    setupDataInfrastructure();
    try {
        c.execute();
        completeTransaction();
    } catch (Exception condition) {
        rollbackTransaction();
        throw condition;
    } finally {
            cleanUp();
        }
    }

    public void addOrderFrom(final ShoppingCart cart, final String userName,
                             final Order order) throws Exception {
        wrapInTransaction(new Command() {
            public void execute() {
                add(order, userKeyBasedOn(userName));
                addLineItemsFrom(cart, order.getOrderKey());
            }
        });
    }
```

In Clojure you have the ability to pass functions around the same as any other value, or if you just need to declare something inline you can leverage anonymous lambda expressions. You can rewrite the previous example in Clojure to look like this code:

```
(defn wrapInTransaction [f]
     (do
          (startTransaction)
          (f)
          (completeTransaction)))

(wrapInTransaction #(
     (do
          (add order user)
          (addLineItemsFrom cart orderKey))))
```

To put it another way, with imperative languages you usually have to be more concerned with how you do things, and in Clojure you're able to focus more on the what you want to do. You can define abstractions at a different level than what is possible in most imperative languages.

Partials

In object-oriented programming there are many patterns for building up objects in steps by using the Builder Pattern, or many related types of objects using the Abstract Factory Pattern. In Clojure, since the primary method of abstraction is the function, you also have a mechanism to build new functions out of existing ones with some of the arguments fixed to a value by using `partial`.

The canonical example of how to use `partial`, shown here is a bit trivial.

```
(def add2 (partial + 2))
```

For a better example, we'll take a look at the `clojure.java.jdbc` library. In the following listing is an example showing a typical pattern for defining the connection properties for your database, and a few simple query wrappers. Notice how every call to `jdbc/query` and `jdbc/insert!` takes the `spec` as its first parameter.

```
(ns sampledb.data
  (:require [clojure.java.jdbc :as jdbc]))

(def spec {:classname "org.postgresql.Driver"
           :subprotocol "postgresql"
           :subname "//localhost:5432/sampledb"})

(defn all-users []
  (jdbc/query spec ["select * from login order by username desc"])))

(defn find-user [username]
  (jdbc/query spec ["select * from login where username = ?" username]))

(defn create-user [username password]
  (jdbc/insert! spec :login {:username username :password password :salt
    "some_salt"}))
```

There is a bit too much repetition in this example, and it only contains three functions for querying the database. Imagine how many times this occurs in a non-trivial application. You can remove this duplication by using `partial`, and creating a new function with this first parameter already bound to the `spec` variable as shown here:

```
(ns sampledb.data
  (:require [clojure.java.jdbc :as jdbc]))

(def spec {:classname "org.postgresql.Driver"
           :subprotocol "postgresql"
           :subname "//localhost:5432/sampledb"})

(def query (partial jdbc/query spec))
(def insert! (partial jdbc/insert! spec))

(defn all-users []
  (query ["select * from login order by username desc"])))

(defn find-user [username]
  (query ["select * from login where username = ?" username]))

(defn create-user [username password]
  (insert! :login {:username username :password password :salt "some_salt"}))
```

Another useful way to use `partial` is for one of the higher order functions such as `map`, which expects a function with exactly one argument to apply to the objects in a collection. You can use `partial` to easily take a function that would normally require more than one argument and create a new one specifying any number of them so that it now only requires one. For example, the `*` function used for multiplying numbers doesn't make much sense with only one argument, but you can use `partial` to specify what you want to multiply each item by as shown here:

```
(defn apply-sales-tax [items]
  ((map (partial * 1.06) items)))
```

The only real downside to `partial` is that you are only able to bind values to parameters in order, meaning that the parameter order is important. If you want to bind the last parameter to a function, you can't leverage `partial`. Instead, you can define another function that wraps the original function call or leverages a lambda expression.

Function Composition

Another useful piece of functionality is the ability to compose multiple functions together to make a new function. Once again, Clojure shows its functional roots based in mathematics. As an example, if you had a function called `f` and another called `g`, you could compose them together such that the output from `f` is fed as the input to `g`, in the same way you can leverage pipes and redirects on the Unix command line and compose several functions together. More specifically, if you have a function call that looks like `(g (f (x))`, you can rewrite it to read as `((comp g f) x)`.

To provide a more practical example, say you wanted to minify some JavaScript, or read in a JavaScript file and remove all of the new lines and extra whitespace, so that it requires less information to transfer from the server to the browser. You can accomplish this task by composing the common string functions provided by Clojure, `str/join`, `str/trim`, `str/split-lines`, as shown here:

```
  (defn minify [input]
   (str/join (map str/trim (str/split-lines input))))
```

This can then be rewritten using the `comp` function to look like the following:

```
(def minify (comp str/join (partial map str/trim) str/split-lines))
```

Notice how the ordering of the functions passed to `comp` retain their original order of the last function being applied first, working your way back to the beginning of the list. Also we modified it a bit to leverage `partial` with the `map` and `str/trim` functions, to create a function that operates on a collection, since `str/trim` only operates on a single string.

Embracing Laziness

Clojure itself is not considered to be a lazy language in the same sense that a language like Haskell is; however, it does provide support for creating and using lazy sequences. In fact, most of the built in functions like `map`, `filter`, and `reduce` generate lazy sequences for you without you probably even knowing it. You can see this here:

```
user> (def result (map (fn [i] (println ".") (inc i)) '[0 1 2 3]))
#'user/result

user> result
.
.
.
.
(1 2 3 4)
```

When you evaluate the first expression, you don't see any output printed to the console. Had this been a non-lazy sequence, you would have seen the output printed to the screen immediately, because it would have evaluated the `println` expression at the time of building the sequence. Instead, the output is not printed until you ask Clojure to show you what is in the result symbol, and it has to fully realize what's inside the sequence. This is exceptionally useful, because the computation inside the function that you pass to `map` may contain some fairly expensive operation, and that expensive operation by itself may not be an issue. Yet, when the operation is executed, the execution of your application can be slowed by several or even hundreds of times.

Another useful example of a lazy sequence in action is when representing an infinite set of numbers. If you have a set of all real numbers, or a set of all prime numbers, you can set all of these numbers in the Fibonacci sequence as shown here.

```
(def fib-seq
     (lazy-cat [1 1] (map + (rest fib-seq) fib-seq)))

(take 10 fib-seq)
-> (1 1 2 3 5 8 13 21 34 55)
```

This sequence is defined using a lazy sequence, and next you will ask Clojure to give you the first 10 numbers in a sequence. In a language that did not support this level of laziness, this type of data modeling would simply not be possible.

Another example of how this can be useful is by infinitely cycling through a finite collection. For example, if you want to assign an ordinal value to every value in a collection for an example group of a list of people into four groups, you can write something similar to the following:

```
(def names '["Christia" "Arline" "Bethann" "Keva" "Arnold" "Germaine"
             "Tanisha" "Jenny" "Erma" "Magdalen" "Carmelia" "Joana"
             "Violeta" "Gianna" "Shad" "Joe" "Justin" "Donella"
             "Raeann" "Karoline"])

user> (mapv #(vector %1 %2) (cycle '[:first :second :third :fourth]) names)
[[:first "Christia"] [:second "Arline"] [:third "Bethann"] [:fourth "Keva"]
  [:first "Arnold"] [:second "Germaine"] [:third "Tanisha"] [:fourth "Jenny"]
  [:first "Erma"] [:second "Magdalen"] [:third "Carmelia"] [:fourth "Joana"]
  [:first "Violeta"] [:second "Gianna"] [:third "Shad"] [:fourth "Joe"]
  [:first "Justin"] [:second "Donella"] [:third "Raeann"] [:fourth "Karoline"]]
```

If you map over multiple collections, you will apply the function provided to the first item in the first collection, the first item in each successive collection, and then the second and so forth, until one of the collections is completely exhausted. So, in order to map the values `:first`, `:second`, `:third`, and `:fourth` repeatedly over all the names, without having to know how many names exist in the collection, you must find a way to cycle over and over repeatedly through the collection. This is what `cycle` and infinite lazy collections excel at.

When You Really Do Need to Mutate

Just because Clojure favors dealing with values doesn't mean you completely do away with mutable state. It just means you greatly limit mutable state, and instead use quarantine in your specific area of code. Clojure provides a few mechanisms to manage mutable state.

Atoms

Using Atoms is the first and simplest mechanism for handling mutable state provided by Clojure. Atoms provide you with a means to manage some shared state in a synchronous, uncoordinated, or independent manner. So, if you need to only manage a single piece of mutable state at a time, then Atoms are the tool you need.

Up to this point, we've primarily focused on values; however, Atoms are defined and used in a different way. Since Atoms represent something that can potentially change, out of necessity they must represent some reference to an immutable structure. An example of how to define an Atom is shown here.

```
user> (def app-state (atom {}))
#'user/app-state
user> app-state
#atom[{} 0x1f5b7bd9]
```

We've defined an Atom containing an empty map with a stored reference in `app-state`. As you can see by the output in the `repl`, the Atom stores a memory location to the map. Right now it doesn't do a whole lot, so let's associate some values into the map.

```
user> (swap! app-state assoc :current-user "Jeremy")
{:current-user "Jeremy"}
user> app-state
#atom[{:current-user "Jeremy"} 0x1f5b7bd9]
user> (swap! app-state assoc :session-id "some-session-id")
{:current-user "Jeremy", :session-id "some-session-id"}
user> app-state
#atom[{:current-user "Jeremy", :session-id "some-session-id"} 0x1f5b7bd9]
```

To modify `app-state`, Clojure provides you with two different functions called `swap!` and `reset!`, both of which atomically modify the value pointed to by the `app-state` reference. The `swap!` function is designed to take a function that will operate on the value stored in the reference, and will swap out the value with the value returned as a result of executing the function. In the preceding example we provided `swap!` with the `assoc` function to associate a new value into the map for a given keyword.

To simply replace the value referenced in `app-state` you can use the `reset!` function, and provide it with a new value to store in the Atom as shown here:

```
user> (reset! app-state {})
{}
user> app-state
#atom[{} 0x1f5b7bd9]
```

You can see that the app-state now references an empty map again.

Now that you know how to store the shared state in your Atom, you may be wondering how you get the values back out. In order to access the state stored in your Atom, use the `deref/@` reader macro as shown here:

```
user> (swap! app-state assoc :current-user "Jeremy" :session-id "some-session-id")
{:current-user "Jeremy", :session-id "some-session-id"}
user> (:current-user @app-state)
"Jeremy"
user> (:session-id @app-state)
```

```
"some-session-id"
user> (:foo @app-state :not-found)
:not-found
```

Once you de-reference your Atom using the `deref/@` reader macro, you can then interact with your `app-state`, just as if it were a map again.

Refs

While Atoms provide you with a means to manage some shared mutable state for a single value, they are limited by the fact that if you need to coordinate changes between multiple objects, such as the classic example of transferring money from one account to another, you need to use transaction references or Refs for short. Transaction references operate similar to how you would expect database transactions to use a concurrency model called Software Transactional Memory, or STM. In fact, Refs fulfill the first three parts required for ACID compliancy: Atomicity, Consistency, and Isolation. Clojure does not concern itself with Durability, however, since the transactions occur in memory.

To illustrate why you can't just use Atoms for coordinated access, consider the following example.

```
user> (def savings (atom {:balance 500}))
#'user/savings
user> (def checking (atom {:balance 250}))
#'user/checking
user> (do
        (swap! checking assoc :balance 700)
        (throw (Exception. "Oops..."))
        (swap! savings assoc :balance 50))
Exception Oops...  user/eval9580 (form-init1334561956148131819.clj:66)
user> (:balance @checking)
700
user> (:balance @savings)
500
```

Here two Atoms called `savings` and `checking` are defined, and we attempt to modify both of them in a `do` block. We are, however, throwing an exception in between updating the two Atoms. This causes our two accounts to get out of sync. Next, let's look at the same example using Refs.

```
user> (def checking (ref {:balance 500}))
#'user/checking
user> (def savings (ref {:balance 250}))
#'user/savings
user> (dosync
        (commute checking assoc :balance 700)
        (throw (Exception. "Oops..."))
        (commute savings assoc :balance 50))
Exception Oops...  user/eval9586/fn--9587 (form-init1334561956148131819.clj:6)
user> (:balance @checking)
500
user> (:balance @savings)
250
```

As you can see, you create Refs and read values out of them similar to how we did that with Atoms earlier. There are a few minor differences, however, in how you update the value stored in the Ref.

We use `commute` rather than `swap!`, and all update operations must perform on the Refs within a `dosync` block.

Nil Punning

If you're at all experienced in Java, you are very familiar with the dreaded `NullPointerException`. It's probably one of the most prolific errors encountered when developing in Java, so much so that the inventor of the Null reference, Tony Hoare, even gave a talk several years ago stating how big of a mistake it was (`http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare`). It seems odd that everything else, with the exception of primitive values, is an Object, except for `null`. This has led to several workarounds in languages, such as `Optional` in Java, null safe object navigation in Groovy, and even in Objective-C you send messages to `nil` and then just happily ignore them.

Clojure, being a Lisp, adopts the philosophy of `nil` punning. Unlike Java, `nil` has a value, and it simply means "no answer." It can also mean different things in different contexts.

When evaluated as a Boolean expression, like many other dynamic typed languages, it will be equivalent to `false`.

```
user> (if nil "true" "false")
"false"
```

`nil` can also be treated like an empty `Seq`. If you call `first` on `nil`, you get a returned `nil`, because there is no first element. If you then call `last` on `nil`, you also unsurprisingly get `nil`. However, don't assume that `nil` is a `Seq,` because when you call `seq` on `nil` you will get a `false` return.

```
user> (first nil)
nil
user> (last nil)
nil
user> (second nil)
nil
user> (seq? nil)
false
```

Unlike many other Lisps, Clojure does not treat empty lists, vectors, and maps as `nil`.

```
user> (if '() "true" "false")
"true"
user> (if '[] "true" "false")
"true"
user> (if '{} "true" "false")
"true"
```

Because `nil` can take on many meanings, you must be mindful to know when `nil` means false and when it means `nil`. For example, when looking for a value in a map, the value for a key can be `nil`. To determine whether or not it exists in the map, you have to return a default value.

```
user> (:foo {:foo nil :bar "baz"})
nil
user> (:foo {:foo nil :bar "baz"} :not-found)
nil
user> (:foo {:bar "baz"} :not-found)
:not-found
```

Unlike Java, `nil` is everywhere and `for` functions return `nil`. Most functions are/should be written to handle a passed in `nil` value. "Using `nil` where it doesn't make sense in Clojure code is *usually* a type error, not a `NullPointerException`, just as using a number as a function is a type error." (`http://www.lispcast.com/nil-punning`)

The Functional Web

It's interesting to see how web programming has evolved over the years. Many different paradigms have come and gone, with some better than others, and yet we haven't seen the end of this evolution. In the early days of the dynamic web back in the 1990s we saw technologies such as CGI, and languages such as Perl, PHP, and ColdFusion come into popularity. Then, with the rise of object-oriented programming, distributed object technologies such as CORBA and EJB rose up, along with object-centric web service technology such as SOAP, as well as object-focused web programming frameworks such as ASP.NET and JSF.

Recent years have seen a shift toward a more RESTful, micro-service based architecture. Nobody uses CORBA anymore, and even SOAP is a dirty word in many circles. Instead, web programming has started to embrace HTTP and its stateless nature and focus on values. Similar to how functional programming has gained popularity because of the rise in number of cores in modern day computers and the necessity of concurrent programming, the web also needs ways to deal with scaling horizontally rather than just vertically.

So what qualities, if any, does web programming in recent years share with functional programming? At its heart, your endpoints can be thought of as functions that take an HTTP request and transform them into an HTTP response. The HTTP protocol itself is also stateless in nature. True, there are things like cookies and sessions, but those merely simulate some sort of state through a shared secret between the client and server. For the most part, the REST endpoints can be thought of as being referentially transparent, which is why caching technologies are so prevalent.

That's not to say there aren't aspects of web programming that are not very functional. Obviously, it would be very difficult to get anything done without modifying some state somewhere. However, it seems like web programming shares as much if not more in common with functional programming than it does with object-oriented programming. In fact, you may find that there's much more opportunity for composability and reuse than with the component-based technologies that have fallen out of favor.

DOING OBJECT-ORIENTED BETTER

Object-oriented programming promised reusable components, and in many ways failed to deliver. Functional programming delivers on this promise where object-oriented programming couldn't. It may surprise you, but Lisp has been doing object-oriented programming since before Java existed. Most object-oriented languages, by definition, define everything as being an object. The problem is that, by forcing everything to fit into this mold of everything being an object, you end up with objects that exist only to "escort" methods, as cleverly explained in the excerpt from Steve Yegge's post *Execution in the Kingdom of Nouns* below.

> *In Javaland, by King Java's royal decree, Verbs are owned by Nouns. But they're not mere pets; no, Verbs in Javaland perform all the chores and manual labor in the entire kingdom. They are, in effect, the kingdom's slaves, or at very least the serfs and indentured servants. The residents of Javaland are quite content with this situation, and are indeed scarcely aware that things could be any different.*
>
> *Verbs in Javaland are responsible for all the work, but as they are held in contempt by all, no Verb is ever permitted to wander about freely. If a Verb is to be seen in public at all, it must be escorted at all times by a Noun.*
>
> *Of course "escort," being a Verb itself, is hardly allowed to run around naked; one must procure a VerbEscorter to facilitate the escorting. But what about "procure" and "facilitate?" As it happens, Facilitators and Procurers are both rather important Nouns whose job is the chaperonement of the lowly Verbs "facilitate" and "procure," via Facilitation and Procurement, respectively.*
>
> *The King, consulting with the Sun God on the matter, has at times threatened to banish entirely all Verbs from the Kingdom of Java. If this should ever to come to pass, the inhabitants would surely need at least one Verb to do all the chores, and the King, who possesses a rather cruel sense of humor, has indicated that his choice would be most assuredly be "execute."*
>
> *The Verb "execute," and its synonymous cousins "run," "start," "go," "justDoIt," "makeItSo," and the like, can perform the work of any other Verb by replacing it with an appropriate Executioner and a call to execute(). Need to wait? Waiter.execute(). Brush your teeth? ToothBrusher(myTeeth).go(). Take out the garbage? TrashDisposalPlanExecutor.doIt(). No Verb is safe; all can be replaced by a Noun on the run.*
>
> `http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html`

At the heart of object-oriented programming is the concept of organizing your programs through creating classes containing the interesting things about a particular object and the things your objects can do. We call these things classes. We can then create more specialized versions of a class through inheritance. The canonical example of this is describing a program that is responsible for describing shapes (see Figure 1-2).

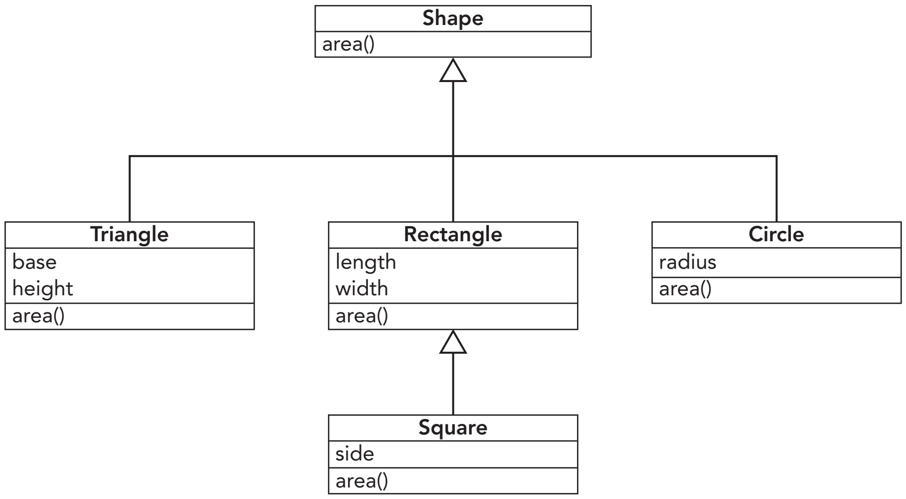

FIGURE 1-2

In the diagram in Figure 1-2 you can see that a generic class of `Shape` is defined, and then several classes that inherit from shape are created, with each defining their own implementation of `area()`. The drawing application sends a message to each of the shapes asking for the area, so at runtime the application determines which implementation to call based on what type of shape it is. This is more commonly known as polymorphism. The key takeaways here are that the behavior belongs to and is defined by the classes themselves, and the methods are invoked on a particular object, and the specific implementation is then decided by the class of the object.

Polymorphic Dispatch with defmulti

Clojure, like many Lisps before it, takes a radically different approach to polymorphism by leveraging a concept called generic functions. This opens up a whole world of possibilities that just were not possible, and for the most part are still not possible, in object-oriented languages. In Clojure, you are not limited to runtime polymorphism on types alone, but also on values, metadata, and relationships between one or more arguments and more.

To rewrite our example above you would start by defining a generic function for `area` as shown here.

```
(defmulti area (fn [shape & _]
                 shape))
```

The generic function consists of first a name for the generic function, then a dispatch function to help Clojure figure out which implementation to call. In this case it's going to inspect the value of the first argument to our function. Then you can implement the various area functions for our different types of shapes like the following.

```
(defmethod area :triangle
  [_ base height]
  (/ (* base height) 2))
```

```
(defmethod area :square
  [_ side]
  (* side side))

(defmethod area :rectangle
  [_ length width]
  (* length width))

(defmethod area :circle
  [_ radius]
  (* radius radius Math/PI))
```

Here you've defined four implementations for the `area` function. You do this by, instead of defining them using `defn`, using the special form `defmethod`, followed by the name of the generic function, and the value from the dispatch function that you would like to match on. Notice how in the parameter lists for each of these functions, you can safely ignore the first parameter being passed in because it was only used for purposes of dispatch. You can see the actual usage of these below.

```
user> (area :square 5)
25
user> (area :triangle 3 4)
6
user> (area :rectangle 4 6)
24
user> (area :circle 5)
78.53981633974483
```

So, you may be wondering how this is any better than what we already have in object-oriented programming. Let's take a look at another example. Suppose you were creating a function that needed to apply a 5% surcharge if a customer lives in New York and a 4.5% surcharge if they live in California. You could model a very simplistic invoice as shown here.

```
{:id 42
 :issue-date 2016-01-01
 :due-date 2016-02-01
 :customer {:name "Foo Bar Industries"
           :address "123 Main St"
           :city "New York"
           :state "NY"
           :zipcode "10101"}
 :amount-due 5000}
```

Writing a similar method that handles this logic in Java would look something like the following.

```
public BigDecimal calculateFinalInvoiceAmount(Invoice invoice) {
     if (invoice.getCustomer().getState().equals("CA")) {
          return invoice.getAmount() * 0.05;
     } else if (invoice.getCustomer.getState().equals("NY")) {
          return invoice.getAmount() * 0.045;
     } else {
          return invoice.getAmount();
     }
}
```

In order to add another state that is needed to add a surcharge for, you must modify this method and add another `else if` conditional. If you wrote this example in Clojure, it would look like the following.

```
(defmulti calculate-final-invoice-amount (fn [invoice]
                                           (get-in invoice [:customer :state])))

(defmethod calculate-final-invoice-amount "CA" [invoice]
  (let [amount-due (:amount-due invoice)]
    (+ amount-due (* amount-due 0.05))))

(defmethod calculate-final-invoice-amount "NY" [invoice]
  (let [amount-due (:amount-due invoice)]
    (+ amount-due (* amount-due 0.045))))

(defmethod calculate-final-invoice-amount :default [invoice]
  (:amount-due invoice))
```

Now, if sometime in the future you decide to add a surcharge to another state, you can simply add another `defmethod` to handle the logic specific to that case.

Defining Types with deftype and defrecord

If you come from a background in object-oriented languages, you may feel compelled to immediately start defining a bunch of custom types to describe your objects, just as you would if you were designing an application in an object-oriented language. Clojure strongly encourages sticking to leveraging the built in types, but sometimes it's beneficial to define the data type so you can leverage things like type-driven polymorphism. In most programs written in object-oriented languages, the classes you define generally fall into one of two categories: classes that do interesting things, and classes that describe interesting things.

For the first of these things Clojure provides you with the `deftype`, and for the latter you can use `defrecord`. Types and records are very similar in how they're defined and used, but there are some subtle differences (see Table 1-1).

TABLE 1-1: deftypes and defrecords

DEFTYPE	DEFRECORD
Supports mutable fields.	Does not support mutable fields.
Provides no functionality other than constructor.	Behaves like a `PersistentMap` and provides default implementations for: ➤ Value based `hashCode` and `equals` ➤ Metadata support ➤ Associative support ➤ Keyword accessors for fields

DEFTYPE	DEFRECORD
Provides reader syntax for instantiating objects using a fully qualified name and argument vector. Passes argument vector directly to constructor. For example: `#my.type[1 2 "a"]`.	Provides additional reader syntax to instantiate objects using a fully qualified name and argument map. For example: `#my.type{:a "foo" :b "bar"}`.
Provides a special function `->YourType`, where `YourType` is the name of your custom type, that passes its arguments to the constructor of your custom type.	Provides a special function, `map->YourRecord`, where `YourRecord` is the name of your custom record, that takes a map and uses it to construct a record from it.

Before looking at how to define and use `deftype` and `defrecord`, you must first look at protocols.

Protocols

If you're at all familiar with Java, you can think of protocols as being very similar to interfaces. They are a named set of functions and their arguments. In fact, Clojure will generate a corresponding Java interface for each of the protocols you define. The generated interface will have methods corresponding to the functions defined in your protocol.

So, let's revisit the Shapes example from earlier. If you create a protocol for Shapes, it looks something like the following.

```
(defprotocol Shape
  (area [this])
  (perimeter [this]))
```

Next, create records for `Square` and `Rectangle` that implement the protocol for Shape as shown here.

```
(defrecord Rectangle [width length]
  Shape
  (area [this] (* (:width this) (:length this)))
  (perimeter [this] (+ (* 2 (:width this)) (* 2 (:length this)))))

(defrecord Square [side]
  Shape
  (area [this] (* (:side this) (:side this)))
  (perimeter [this] (+ (* 4 (:side this)))))
```

Then create and call the functions to calculate the area as shown here.

```
user> (def sq1 (->Square 4))
#'user/sq1
user> (area sq1)
16
user> (def rect1 (->Rectangle 4 2))
```

```
#'user/rect1
user> (area rect1)
8
```

Alternatively, you can also create your records using the `map->Rectangle` and `map->Square` functions as shown here.

```
user> (def sq2 (map->Square {:side 3}))
#'user/sq2
user> (def rect2 (map->Rectangle {:width 4 :length 7}))
#'user/rect2
user> (into {} rect2)
{:width 4, :length 7}
user> rect2
#user.Rectangle{:width 4, :length 7}
user> (into {} rect2)
{:width 4, :length 7}
user> (:width rect2)
4
user> (:length rect2)
7
user> (:foo rect2 :not-found)
:not-found
```

Also, recall that earlier records were discussed, which are basically wrappers around `PersistentMap`. This of course means that you can interact with your records as if they were maps in Clojure. You can access the members of your `Rectangle` object just as if it were a map, and you can even construct new maps from it using `into`.

Reify

Sometimes you want to implement a protocol without having to go through the trouble of defining a custom type or record. For that, Clojure provides `reify`. A quick example of this can be seen here.

```
(def some-shape
  (reify Shape
    (area [this] "I calculate area")
    (perimeter [this] "I calculate perimeter")))

user> some-shape
#object[user$reify__8615 0x221f1bd "user$reify__8615@221f1bd"]
user> (area some-shape)
"I calculate area"
user> (perimeter some-shape)
"I calculate perimeter"
```

You can think of `reify` as the Clojure equivalent of doing anonymous inner classes in Java. In fact, you can use `reify` to create anonymous objects that extend Java interfaces as well.

PERSISTENT DATA STRUCTURES

With most imperative languages, the data structures you use are destructive by nature, replacing values in place. This becomes problematic because if you use destructive data structures, you cannot long pass them around with the confidence that nothing else has come along and modified the values.

For example, if you update the second index of the list L1, shown above, in a language such as Java, you see that the value is updated in place and the list L1 is no longer the same list as before. So anything that may have been using L1 for calculations will have changed as well (see Figure 1-3).

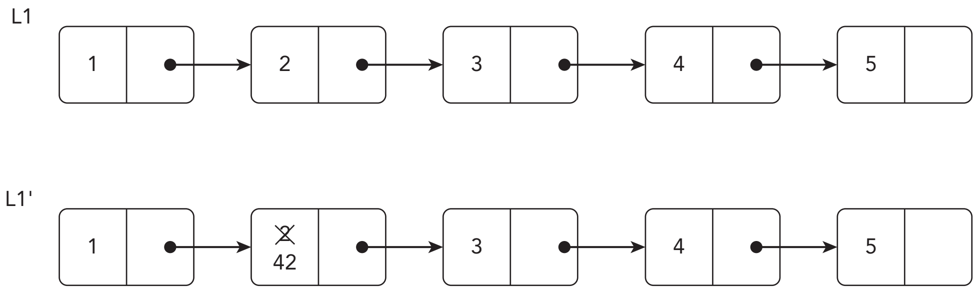

FIGURE 1-3

Clojure, on the other hand, continues with its tradition of focusing on values, and implements many of its data collections in a persistent manner, meaning that any time you do something to modify a collection, it returns a shiny new collection that may also share some of its structure with the original. You may be thinking to yourself that sharing elements between structures like this would be problematic, but because the elements themselves are immutable, you don't need to be concerned.

The example in Figure 1-4 shows a similar update using a persistent list data structure. Notice how when you update the second index in the list L1, you instead create a new list L2, thus creating new copies of all the nodes up to the point where we're updating. This then shares structure with the rest of the original list.

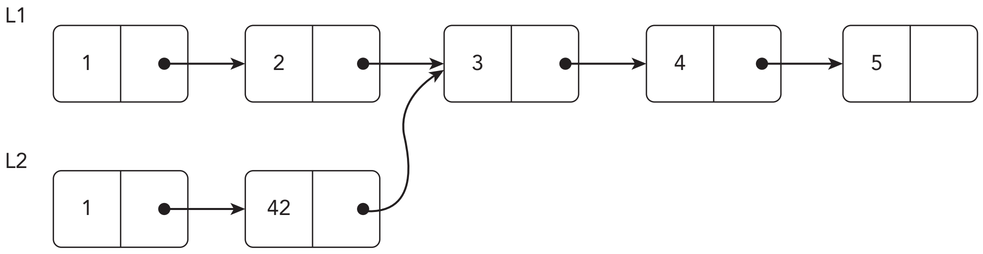

FIGURE 1-4

If you were to append a list L2 to the end of L1, it would be returned L3, which is basically a copy of L1 with the exception of the last node that then points to the beginning of L2. This maintains the integrity of both L1 and L2, so any function that was previously using them can continue to do so without worrying about the data being changed out from under them.

Let's take a look at another example (see Figure 1-5), but this time we'll look at a simplistic binary search tree. If you start with a tree L1 and attempt to add the value 6 to the tree, you see that copies of all the nodes are made containing the path to where you want to add the new node. Then in the new root node (L2), you simply point to the right sub-tree from L1.

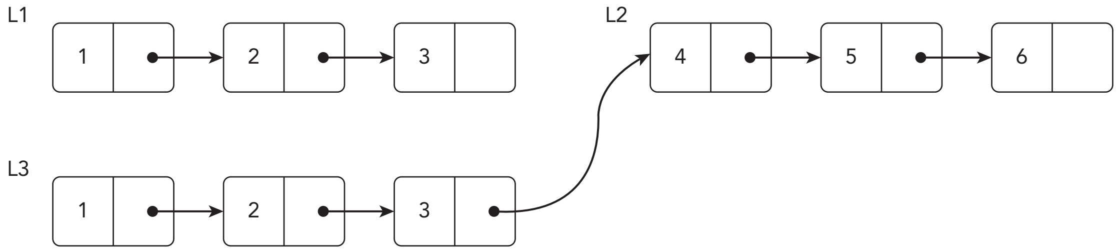

FIGURE 1-5

Next, let's take a look at how to implement the binary tree in Clojure. Start by defining a protocol called `INode` as shown here.

```
(defprotocol INode
  (entry [_])
  (left [_])
  (right [_])
  (contains-value? [_ _])
  (insert-value [_ _]))
```

Let's examine a few pieces of functionality for the example. Let's look at functions that retrieve the left and right sub-trees, regardless of whether the value exists in our tree, with the ability to add new values to the tree. Once you have the protocol defined, you can begin to implement the binary search tree by defining a new type using `deftype` as shown below.

```
(deftype Node [value left-branch right-branch]
  INode
  (entry [_] value)
  (left [_] left-branch)
  (right [_] right-branch)
  (contains-value? [tree v]
    (cond
      (nil? tree) false
      (= v value) true
      (< v value) (contains-value? left-branch v)
      (> v value) (contains-value? right-branch v)))
  (insert-value [tree v]
    (cond
      (nil? tree) (Node. v nil nil)
      (= v value) tree
      (< v value) (Node. value (insert-value left-branch v) right-branch)
      (> v value) (Node. value left-branch (insert-value right-branch v)))))
```

So let's try out the new code.

```
user> (def root (Node. 7 nil nil))
#'user/root
user> (left root)
nil
user> (right root)
nil
user> (entry root)
7
user> (contains-value? root 7)
true
```

So far so good. Now let's see if the tree contains 5.

```
user> (contains-value? root 5)
IllegalArgumentException No implementation of method: :contains-value? of protocol:
   #'user/INode found for class: nil  clojure.core/-cache-protocol-fn
  (core_deftype.clj:554)
```

What happened? If you investigate the error message it's trying to tell you that `nil` doesn't implement the protocol and thus it doesn't know how to call the function `contains-value?` on `nil`. You can fix this by extending the protocol onto `nil` as shown here.

```
(extend-protocol INode
  nil
  (entry [_] nil)
  (left [_] nil)
  (right [_] nil)
  (contains-value? [_ _] false)
  (insert-value [_ value] (Node. value nil nil)))
```

This now allows you to refactor the Node type to remove the redundant checks for `nil` to look like the following.

```
(deftype Node [value left-branch right-branch]
  INode
  (entry [_] value)
  (left [_] left-branch)
  (right [_] right-branch)
  (contains-value? [tree v]
    (cond
      (= v value) true
      (< v value) (contains-value? left-branch v)
      (> v value) (contains-value? right-branch v)))
  (insert-value [tree v]
    (cond
      (= v value) tree
      (< v value) (Node. value (insert-value left-branch v) right-branch)
      (> v value) (Node. value left-branch (insert-value right-branch v)))))
```

Now that we have that fixed, let's try this out again.

```
user> (contains-value? root 5)
false
```

Excellent. Now let's create a tree with a few more nodes.

```
user> (def root (Node. 7 (Node. 5 (Node. 3 nil nil) nil) (Node. 12
 (Node. 9 nil nil) (Node. 17 nil nil))))
#'user/root
```

The above code should produce a tree with the same structure as L1 shown in Figure 1-6.

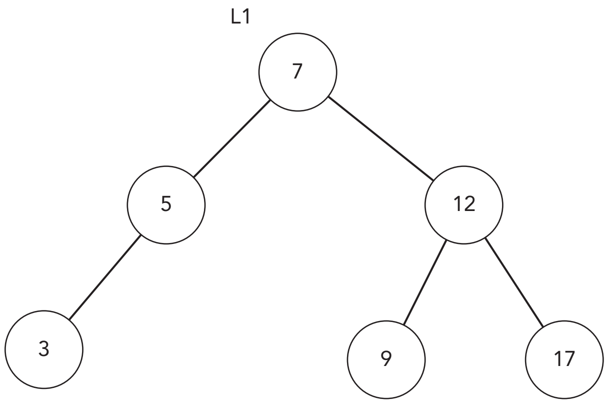

FIGURE 1-6

You can validate that assumption with the following commands.

```
user> (left root)
#object[user.Node 0x5cedcfe8 "user.Node@5cedcfe8"]
user> (entry (left root))
5
user> (entry (left (left root)))
3
user> (entry (right root))
12
user> (entry (right (right root)))
17
```

As you can see, when you ask for the value of the left sub-tree from root, you get the value 5, and when you ask for the left of that sub-tree, you get the value 3. Now, let's take a look at the identity value for the left and right sub-trees from `root` respectively.

```
user> (identity (left root))
#object[user.Node 0x5cedcfe8 "user.Node@5cedcfe8"]
user> (identity (right root))
#object[user.Node 0x124ee325 "user.Node@124ee325"]
```

Your values may differ slightly from above, but they should look similar. Next, let's add a new value of 6 to the tree, which should be inserted to the right of the 5 node. After you insert the new value, take a look at the identity values again from the root node of the new tree you just created.

```
user> (def l (insert-value root 6))
#'user/l
```

```
user> (identity (left l))
#object[user.Node 0x167286ec "user.Node@167286ec"]
user> (identity (right l))
#object[user.Node 0x124ee325 "user.Node@124ee325"]
```

You should see that a new Node for the left sub-tree of our tree is created, but the new list is pointing at the same instance of the right sub-tree as the original tree did. The result of the inserts should now produce the structure shown in Figure 1-7. With the original list still intact, the new list shares some structure with the original.

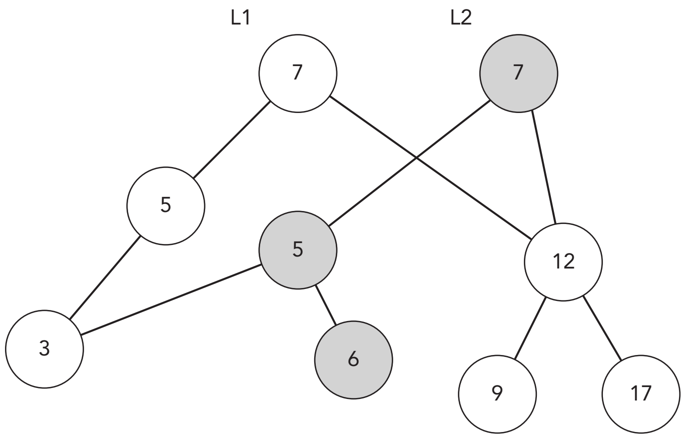

FIGURE 1-7

If you want to read more about the way that Clojure implements some of its persistent data structures, there are a pair of great articles explaining the implementation details found at `http://blog.higher-order.net/2009/02/01/understanding-clojures-persistentvector-implementation` and `http://blog.higher-order.net/2009/09/08/understanding-clojures-persistenthashmap-deftwice`.

SHAPING THE LANGUAGE

You may have heard at one point in time Lisp being described as a "programmable programming language," and that Lisp is homoiconic, or even how "code is data, data is code." What does this really mean for you as a programmer though? If you have a background in C, you may be familiar with the term "macro"; however, as stated earlier in this chapter, forget everything you think you know about macros. Macros in Clojure are a much more powerful construct than what is available in any imperative language. This is so powerful that entire books have been written on macros alone (see *Let Over Lambda* and *Mastering Clojure Macros*).

Macros in Clojure allow you to rewrite the normal syntax rules of Clojure in order to shape the language to fit your problem domain. While some languages offer some sort of mechanism to do metaprogramming, and the ability to modify default behavior, none of them exhibit quite the power of Clojure's macro system. Many frameworks in other languages seem to abuse this power, and this often leads to tricky to find bugs and confusion about where certain functionality comes from. So, you must exercise caution when deciding whether or not to use a macro.

So, what exactly constitutes a good use of macros and what doesn't? One exemplary example of how you can leverage macros to create a powerful and expressive DSL is the routes library in the Compojure framework. The routes library defines macros for creating ring handler mappings using a grammar that make perfect sense in this context. We see an example of the `defroutes` macro coupled with the various macros that map to the HTTP verbs:

```
(defroutes app-routes
 (GET "/" [] (index))
 (GET "/books" [] (get-books))
 (GET "/books/:id" [id] (find-book id))
 (POST "/books" [title author] (add-book title author))
```

By leveraging macros, Compojure is able to change the semantics of how the language works. It allows you to define the mapping between URL and handler in a more natural fashion. The `defroutes` macro allows you to create a named set of routes, in this case called `app-routes`. Then a list of routes is provided that is evaluated by Ring until it finds one that matches. The routes themselves are defined using a macro that allows you to specify the HTTP verb, followed by the route, with the ability to define path variable bindings. Next we list any variables that will be bound. These can come from URL parameters, or in the case of the POST route, from the form parameters in the request. Finally, you are able to either define the actual handler inline, if it's simple enough, or have the route dispatch to a function defined elsewhere.

Another fine example of the power of macros and how you can build a natural fluent API is the Honey SQL library found at `https://github.com/jkk/honeysql`. It allows you to define SQL queries using Clojure's built in data structures and then provides a multitude of functions and macros that transform them into `clojure.java.jdbc`, and compatible parameterized SQL that you can then pass directly to `jdbc/query`, `jdbc/insert!`, and the like. Let's take a look at one of the examples from their documentation.

```
(def sqlmap {:select [:a :b :c]
             :from [:foo]
             :where [:= :f.a "baz"]})
(sql/format sqlmap)

=> ["SELECT a, b, c FROM foo WHERE (f.a = ?)" "baz"]
```

Honey SQL even defines a helper function called `build` that helps you define these map objects as shown below.

```
(sql/build :select :*
           :from :foo
           :where [:= :f.a "baz"])

=> {:where [:= :f.a "baz"], :from [:foo], :select [:*]}
```

Leveraging the build function allows you to still use the same style of specifying the query; however, it doesn't require all of the extra brackets as before, making it just that much more concise.

Clojure also offers some really nice metaprogramming abilities through `defprotocol`. Protocols are similar to Java interfaces in that they define a specific set of functions and their signatures, and even generate a corresponding interface that you can use in Java, which we'll cover later. The other thing you can do with protocols, though, is extend existing types, including final classes in Java. This means that you can add new methods to things like Java's `String` class as shown here.

```
(defprotocol Palindrome (is-palindrome? [object]))

(extend-type java.lang.String
  Palindrome
  (is-palindrome? [s]
    (= s (apply str (reverse s)))))

(is-palindrome? "tacocat")

=> true
```

You can see how to define a protocol called `Palindrome`, and define it as having a single function called `is-palindrome?`. Next, extend the `java.lang.String` class to add functionality to Java's built-in String class. Then, show it in action by calling `is-palindrome?` with the value `"tacocat"`.

As mentioned before, this level of modifying the language and types should be carefully considered before you decide to use it. It often leads to the same problems mentioned before with overuse of metaprogramming facilities and a lack of clarity about where things may be defined, especially when you can get by with just defining a regular function.

Clojure offers some very powerful ways to shape the language to the way you want to work. How you decide to use it is mostly a matter of personal preference.

SUMMARY

In this chapter you have seen how Clojure is different than most other mainstream languages in use today. As defined at the beginning of the chapter, if you don't come at it with a clear mind and learn how to do things the Clojure way, you'll simply be writing the same old code in a different syntax. Enlightenment will not come overnight; however, if you do approach it with an open mind it will likely fundamentally change the way you think about programming in general. As stated by Eric Raymond, "Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot."