SAMPLE CHAPTER

THE Joy OF Clojure

Michael Fogus
Chris Houser

FOREWORD BY STEVE YEGGE

MANNING

The Joy of Clojure
by Michael Fogus and Chris Houser

Chapter 9

brief contents

9 Combining data and code

This chapter covers

- Namespaces
- Exploring Clojure multimethods with the Universal Design Pattern
- Types, protocols, and records
- Putting it all together: a fluent builder for chess moves

Clojure provides powerful features for grouping and partitioning logical units of code and data. Most logical groupings occur within namespaces, Clojure's analogue to Java packages. We explore how to build, manipulate, and reason about them. Also, in this chapter we'll play with Clojure's powerful multimethods that provide polymorphism based on arbitrary dispatch functions. We then uncover recent additions to Clojure supporting *abstraction-oriented programming*—types, protocols, and records. Finally, the chapter concludes with the creation of a fluent chess-move facility, comparing a Java approach to solving the problem with a Clojure approach.

9.1 Namespaces

Newcomers to Clojure have a propensity to hack away at namespace declarations until they appear to work. This may work sometimes, but it delays the process of learning how to leverage namespaces more effectively.

From a high-level perspective, namespaces can be likened to a two-level mapping, where the first level is a symbol to a namespace containing mappings of symbols to Vars, as shown in figure 9.1. This conceptual model[1] is slightly complicated by the fact that namespaces can be aliased, but even in these circumstances the model holds true.

In the simplest possible terms, qualified symbols of the form `joy.ns/authors` cause a two-level lookup: a symbol `joy.ns` used to lookup a namespace map and a symbol `authors` used to retrieve a Var, as shown in the following listing.

Listing 9.1 Namespace navigation

```
(in-ns 'joy.ns)
(def authors ["Chouser"])

(in-ns 'your.ns)
(clojure.core/refer 'joy.ns)
joy.ns/authors
;=> ["Chouser"]

(in-ns 'joy.ns)
(def authors ["Chouser" "Fogus"])

(in-ns 'your.ns)
joy.ns/authors
;=> ["Chouser" "Fogus"]
```

Because a symbolic name refers to a Var in the current namespace or another, it follows that any referred Var always evaluates to the current value and not the value present at referral time.

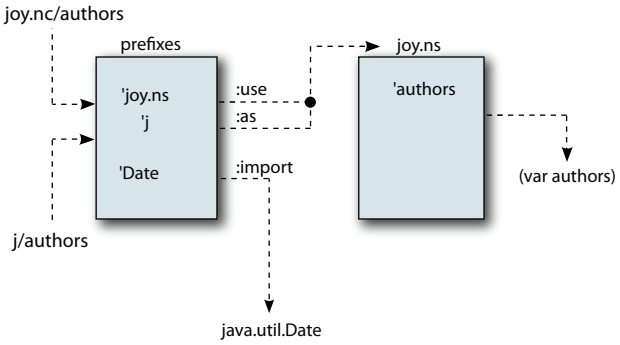

Figure 9.1 The logical layout of namespaces. The process to resolve a Var `joy.ns/authors` includes a symbolic resolution of the namespace and the Var name. The result is the Var itself. Aliases created with `:use` work as expected.

[1] As always, we're trying to keep the level of discussion limited to abstractions rather than implementation details.

9.1.1 Creating namespaces

There are a number of ways to create a new namespace; each has its advantages and use cases. The choice of one namespace-creation mechanism over another amounts to choosing the level of control over the default symbolic mappings.

NS

In idiomatic Clojure source code, you'll see the `ns` macro used almost exclusively. By using the `ns` macro, you automatically get two sets of symbolic mappings—all classes in the `java.lang` package and all of the functions, macros, and special forms in the `clojure.core` namespace:

```
(ns chimp)
(reduce + [1 2 (Integer. 3)])
;=> 6
```

Using the `ns` macro creates a namespace if it doesn't already exist, and switches to that namespace. The `ns` macro is intended for use in source code files and not in the REPL, although there's nothing preventing it.

IN-NS

Using the `in-ns` function also imports the `java.lang` package like `ns`; but it doesn't create any mappings for functions or macros in `clojure.core`. The `in-ns` function also takes an explicit symbol used as the namespace qualifier, as in

```
(in-ns 'gibbon)

(reduce + [1 2 (Integer. 3)])
; java.lang.Exception: Unable to resolve symbol: reduce in this context

(clojure.core/refer 'clojure.core)
(reduce + [1 2 (Integer. 3)])
;=> 6
```

The `in-ns` function is more amenable to REPL experimentation when dealing with namespaces than `ns`.

CREATE-NS

The finest level of control for creating namespaces is provided through the `create-ns` function, which when called takes a symbol and returns a namespace object:

```
(def b (create-ns 'bonobo))
b
;=> #<Namespace bonobo>

((ns-map b) 'String)
;=> java.lang.String
```

The call to `create-ns` doesn't switch to the named namespace, but it does create Java class mappings automatically. When given a namespace object (retrieved using the `find-ns` function also), you can manipulate its bindings programmatically using the functions `intern` and `ns-unmap`:

```
(intern b 'x 9)
;=> #'bonobo/x
bonobo/x
;=> 9
```

In the preceding code, we bound the symbol `x` to the value `9` in the namespace `bonobo`, and then referenced it directly using its qualified name `bonobo/x`. We can do the same thing for any type of Var binding:

```
(intern b 'reduce clojure.core/reduce)
;=> #'bonobo/reduce

(intern b '+ clojure.core/+)
;=> #'bonobo/+

(in-ns 'bonobo)
(reduce + [1 2 3 4 5])
;=> 15
```

Because only Java class mappings are created by `create-ns`, you'll have to `intern` any Clojure core functions, as we did with `+` and `reduce`. You can even inspect the mappings within a namespace programmatically, and likewise remove specific mappings:

```
(in-ns 'user)
(get (ns-map 'bonobo) 'reduce)
;=> #'bonobo/reduce

(ns-unmap 'bonobo 'reduce)  ;=> nil

(get (ns-map 'bonobo) 'reduce)
;=> nil
```

Finally, you can wipe a namespace using `remove-ns`:

```
(remove-ns 'bonobo)
;=> #<Namespace bonobo>

(all-ns)
;=> (#<Namespace clojure.set> #<Namespace clojure.main>
     #<Namespace clojure.core> #<Namespace clojure.zip>
     #<Namespace chimp> #<Namespace gibbon>
     #<Namespace clojure.xml>)
```

You should be careful when populating namespaces using `create-ns` and `intern`, because they cause potentially confusing side-effects to occur. Their use is intended only for advanced techniques, and even then they should be used cautiously.

9.1.2 *Expose only what's needed*

Knowing that namespaces operate as a two-level mapping will only get you so far in creating and using them effectively. You must understand other practical matters to use namespaces to their fullest. For example, for a given namespace `joy.contracts`, your directory structure could look like that in figure 9.2.

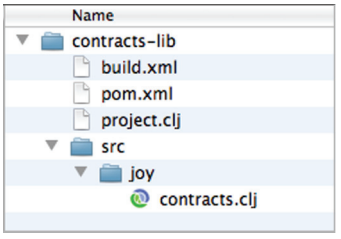

Figure 9.2 Namespace private directories: the directories layout for an illustrative `joy.contracts` namespace

This directory structure is fairly straightforward, but there are a couple items to note. First, though the namespace is named `joy.contracts`, the corresponding Clojure source file is located in the contracts-lib/src/joy directory. This is a common technique in organizing Java and Clojure projects, where the actual source directories and files are located in a common src subdirectory in the main project directory. The additional files build.xml, pom.xml, and project.clj correspond to the build scripts for Apache Ant, Maven, and Leiningen, respectively. These build scripts will know, through either configuration or convention, that the src directory contains the directories and source files for Clojure namespaces and *not* part of the namespace logical layout. If you were to open the contracts.clj file located in contracts-lib/src/joy in your favorite editor, then you might see something like that shown in figure 9.3.

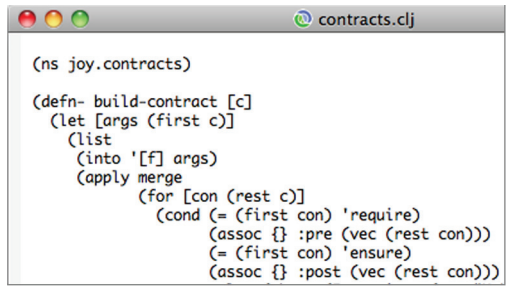

Figure 9.3 Namespace private source: the top of the source file for the `joy.contracts` namespace

The file contracts.clj defines the namespace `joy.contracts` and defines the function `build-contract` using the `defn-` macro. The use of `defn-` in this way indicates to Clojure that the `build-contract` function is private to the `joy.contracts` namespace. The `defn-` macro is provided for convenience and simply attaches privileged metadata to the Var containing the function. You could attach the same namespace privacy metadata yourself, as shown:

```
(ns hider.ns)

(defn ^{:private true} answer [] 42)

(ns seeker.ns
  (:refer hider.ns))

(answer)
; java.lang.Exception: Unable to resolve symbol: answer in this context
```

The use of `^{:private true}` in this way will also work within a `def` and a `defmacro`, and for these cases it's required, because there's no corresponding `def-` and `defmacro-` in Clojure's core.

> **HYPHENS/UNDERSCORES** If you decide to name your namespaces with hyphens, à la `my-cool-lib`, then the corresponding source file *must* be named with underscores in place of the hyphens (my_cool_lib.clj).

Because Clojure namespace names are tied to the directory in which they reside, you can also create a certain directory structure conducive to hiding implementation details, as seen in figure 9.4.

By creating another subdirectory to contracts-lib/src/joy named impl, you can effectively hide implementation details for your code. The public-facing API would be

located in contracts.clj and the "hidden" implementation details in impl.clj. Your clients would be expected to refer only to the elements in contracts.clj, whereas your library could refer to elements in impl.clj, as shown in figure 9.5.

Of course, nothing's stopping you from also referencing the `joy.contracts.impl` namespace, but you do so at their own peril. There are never any guarantees that implementation details will remain the same shape from one release to the next.

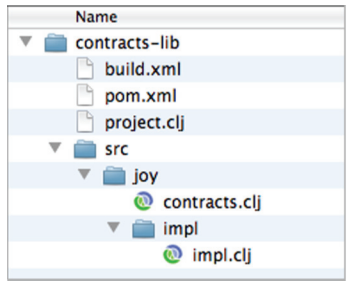

Figure 9.4 Private API directories: using the folder layout to hide namespace implementation details

9.1.3 Declarative inclusions and exclusions

When defining namespaces, it's important to include only the references that are likely to be used. Clojure prefers a fine-grained Var mapping via a set of directives on the `ns` macro: `:exclude`, `:only`, `:as`, `:refer-clojure`, `:import`, `:use`, `:load`, and `:require`.

We'll describe a namespace named `joy.ns-ex` first in prose and then using `ns` and its directives. In this namespace, we want to exclude the `defstruct` macro from

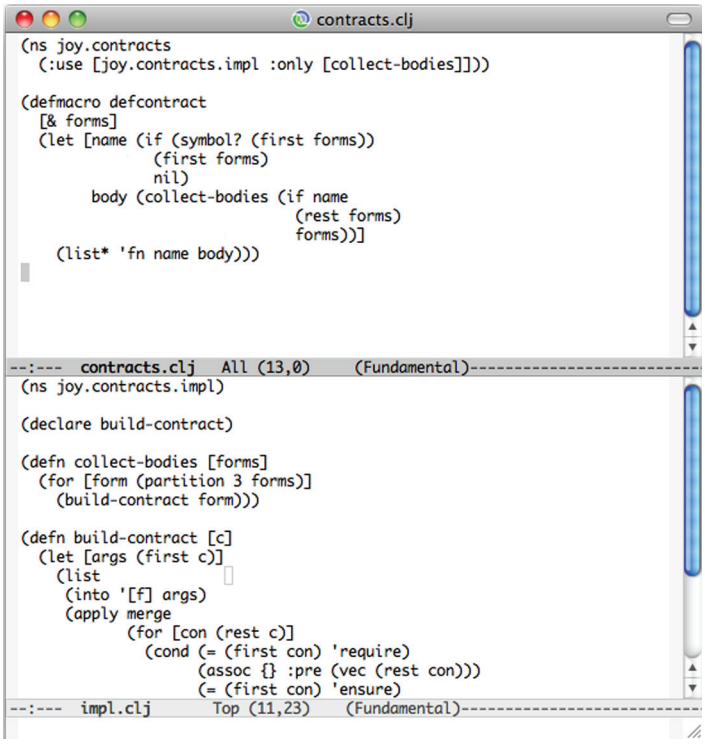

Figure 9.5 Private API source: the client-facing API is located in contracts.clj and the private API in impl.clj.

`clojure.core`. Next, we want to use everything in `clojure.set` and `clojure.xml` without namespace qualification. Likewise, we wish to use only the functions `are` and `is` from the `clojure.test` namespace without qualification. We then want to load the `clojure.zip` namespace and alias it as `z`. Finally, we want to import the Java classes `java.util.Date` and `java.io.File`. By providing directives, the problem of namespace inclusions and exclusions become a declarative matter, as shown:

```
(ns joy.ns-ex
  (:refer-clojure :exclude [defstruct])
  (:use (clojure set xml))
  (:use [clojure.test :only (are is)])
  (:require (clojure [zip :as z]))
  (:import (java.util Date)
           (java.io File)))
```

We'll touch on further uses of namespaces throughout the rest of the book, with an extensive example explaining their use as JVM class specifications in section 10.3.

> **AVOID NAKED :USE** One point of note that we should mention is that the `(:use (clojure set xml))` statement is considered a promiscuous operation and therefore discouraged. The `:use` directive without the `:only` option pulls in all of the public Vars in `clojure.set` and `clojure.xml` indiscriminately. Though this practice is useful when incrementally building your code, it shouldn't endure into the production environment. When organizing your code along namespaces, it's good practice to export and import *only* those elements needed.

We now turn our focus to Clojure's multimethods, a way of defining polymorphic functions based on the results of arbitrary functions, which will get you halfway toward a system of polymorphic types.

9.2 Exploring Clojure multimethods with the Universal Design Pattern

The most specific event can serve as a general example of a class of events.

—Douglas R. Hofstadter

In Douglas Hofstadter's Pulitzer prize winning work *Gödel, Escher, Bach: An Eternal Golden Braid*, he describes a notion of the *Prototype Principle*—the tendency of the human mind to use specific events as models for similar but different events or things. He presents the idea "that there is generality in the specific" (Hofstadter 1979). Building on this idea, programmer Steve Yegge coined the term *The Universal Design Pattern (UDP)*, extrapolating on Hofstadter's idea (Yegge 2008) and presenting it in terms of prototypal inheritance (Ungar 1987).

The UDP is built on the notion of a map or map-like object. Though not groundbreaking, the flexibility in the UDP derives from the fact that each map contains a reference to a *prototype* map used as a parent link to inherited fields. You might wonder how anyone could model a software problem in this way, but we assure you that

countless programmers do so every day when they choose JavaScript (Flanagan 2006). In this section, we'll implement a subset of Yegge's UDP and discuss how it might be used as the basis for abstraction-oriented programming and polymorphism using Clojure's multimethods and ad hoc hierarchies.

9.2.1 The parts

In addition to the aforementioned prototype reference, the UDP requires a set of supporting functions to operate: `beget`, `get`, `put`, `has?`, and `forget`. The entire UDP is built on these five functions, but we'll need the first three for this section.

BEGET

The `beget` function performs a simple task. It takes a map and associates its prototype reference to another map, returning a new map:

```
(ns joy.udp
  (:refer-clojure :exclude [get]))

(defn beget [o p] (assoc o ::prototype p))

(beget {:sub 0} {:super 1})
;=> {:joy.udp/prototype {:super 1}, :sub 0}
```

To participate in the UDP, maps must have a `:joy.udp/prototype` entry.

PUT

The function `put` takes a key and an associated value and puts them into the supplied map, overwriting any existing key of the same name:

```
(def put assoc)
```

The `put` function is asymmetric to the functionality of `get`. The `get` method retrieves values anywhere along the prototype chain, whereas `put` only ever inserts at the level of the supplied map.

GET

Because of the presence of the prototype link, `get` requires more than a simple one-level lookup. Instead, whenever a value isn't found in a given map, the prototype chain is followed until the end:

```
(defn get [m k]
  (when m
    (if-let [[_ v] (find m k)]
      v
      (recur (::prototype m) k))))

(get (beget {:sub 0} {:super 1})
     :super)
;=> 1
```

We don't explicitly handle the case of "removed "properties, but instead treat them like any other associated value. This is fine because the "not found" value of `nil` is falsey. Most of the time, it's sufficient to rely on the fact that looking up a nonexistent

key will return `nil`. But in cases where you want to allow users of your functions to store any value at all, including `nil`, you'll have to be careful to distinguish `nil` from "not found," and the `find` function is the best way to do this.

9.2.2 Usage

Using only `beget`, `put`, and `get`, you can leverage the UDP in some simple, yet powerful ways. Assume that at birth cats like dogs and only learn to despise them when goaded. Morris the cat has spent most of his life liking 9-Lives cat food and dogs, until the day comes when a surly Shih Tzu leaves him battered and bruised. We can model this unfortunate story as shown:

```
(def cat {:likes-dogs true, :ocd-bathing true})
(def morris (beget {:likes-9lives true} cat))
(def post-traumatic-morris (beget {:likes-dogs nil} morris))

(get cat :likes-dogs)
;=> true

(get morris :likes-dogs)
;=> true

(get post-traumatic-morris :likes-dogs)
;=> nil
```

The map `post-traumatic-morris` is like the old `morris` in every way except for the fact that he has learned to hate dogs. Modeling cat and dog societal woes is interesting but far from the only use case for the UDP, as you'll see next.

NO NOTION OF SELF

Our implementation of the UDP contains no notion of self-awareness via an implicit `this` or `self` reference. Though adding such a feature would probably be possible, we've intentionally excluded it in order to draw a clear separation between the prototypes and the functions that work on them (Keene 1989). A better solution, and one that follows in line with a deeper Clojure philosophy, would be to access, use, and manipulate these prototypes using Clojure's multimethods.

9.2.3 Multimethods to the rescue

Adding behaviors to the UDP can be accomplished easily using Clojure's multimethod facilities. *Multimethods* provide a way to perform function polymorphism based on the result of an arbitrary dispatch function. Coupled with our earlier UDP implementation, we can implement a prototypal object system with differential inheritance similar to (although not as elegant as) that in the Io language (Dekorte Io). First, we'll need to define a multimethod `compiler` that dispatches on a key `:os`:

```
(defmulti  compiler :os)
(defmethod compiler ::unix [m] (get m :c-compiler))
(defmethod compiler ::osx  [m] (get m :c-compiler))
```

The multimethod `compiler` describes a simple scenario: if the function `compiler` is called with a prototype map, then the map is queried for an element `:os`, which has methods defined on the results for either `::unix` or `::osx`. We'll create some prototype maps to exercise `compiler`:

```
(def clone (partial beget {}))
(def unix   {:os ::unix, :c-compiler "cc", :home "/home", :dev "/dev"})
(def osx  (-> (clone unix)
              (put :os ::osx)
              (put :c-compiler "gcc")
              (put :home "/Users")))

(compiler unix)
;=> "cc"

(compiler osx)
;=> "gcc"
```

That's all there is (Foote 2003) to creating behaviors that work against the specific "type" of a prototype map. But a problem of inherited behaviors still persists. Because our implementation of the UDP separates state from behavior, there's seemingly no way to associate inherited behaviors. But as we'll now show, Clojure does provide a way to define ad hoc hierarchies that we can use to simulate inheritance within our model.

9.2.4 Ad hoc hierarchies for inherited behaviors

Based on the layout of the `unix` and `osx` prototype maps, the property `:home` is overridden in `osx`. We could again duplicate the use of `get` within each method defined (as in `compiler`), but instead we prefer to say that the lookup of `:home` should be a derived function:

```
(defmulti home :os)
(defmethod home ::unix [m] (get m :home))

(home unix)
;=> "/home"

(home osx)
; java.lang.IllegalArgumentException:
;   No method in multimethod 'home' for dispatch value: :user/osx
```

Clojure allows you to define a relationship stating "`::osx` is a `::unix`" and have the derived function take over the lookup behavior using Clojure's `derive` function:

```
(derive ::osx ::unix)
```

Now the `home` function works:

```
(home osx)
;=> "/Users"
```

You can query the derivation hierarchy using the functions `parents`, `ancestors`, `descendants`, and `isa?` as shown:

```
(parents ::osx)
;=> #{:user/unix}
```

```
(ancestors ::osx)
;=> #{:user/unix}

(descendants ::unix)
;=> #{:user/osx}

(isa? ::osx ::unix)
;=> true
(isa? ::unix ::osx)
;=> false
```

The result of the `isa?` function defines how multimethods dispatch. In the absence of a derivation hierarchy, `isa?` can be likened to pure equality, but with it traverses a derivation graph.

9.2.5 Resolving conflict in hierarchies

What if we interject another ancestor into the hierarchy for `::osx` and want to again call the `home` method? Observe the following:

```
(derive ::osx ::bsd)
(defmethod home ::bsd [m] "/home")

(home osx)
; java.lang.IllegalArgumentException: Multiple methods in multimethod
;  'home' match dispatch value: :user/osx -> :user/unix and
;  :user/bsd, and neither is preferred
```

As shown in figure 9.6, `::osx` derives from both `::bsd` and `::unix`, so there's no way to decide which method to dispatch, because they're both at the same level in the derivation hierarchy. Fortunately, Clojure provides a way to assign favor to one method over another using the function `prefer-method`:

```
(prefer-method home ::unix ::bsd)
(home osx)
;=> "/Users"
```

In this case, we used `prefer-method` to explicitly state that for the multimethod `home`, we prefer the method associated with the dispatch value `::unix` over the one for `::bsd`, as illustrated in figure 9.5. As you'll recall, the `home` method for `::unix` explicitly used `get` to traverse the prototype chain, which is the preferred behavior.

As you might expect, removing the `home` method for the `::bsd` dispatch value using `remove-method` will remove the preferential lookup for `::osx`:

```
(remove-method home ::bsd)
(home osx)
;=> "/Users"
```

All of these functions manipulate and operate off of the global hierarchy map directly. If you prefer to reduce these potentially confusing side-effects, then you can define a derivation hierarchy using `make-hierarchy` and `derive`:

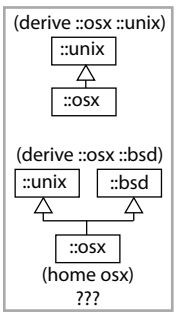

Figure 9.6 Hierarchy conflict: most languages allowing type derivations use a built-in conflict-resolution strategy. In the case of CLOS, it's fully customizable. Clojure requires conflicts to be resolved with `prefer-method`.

```
(derive (make-hierarchy) ::osx ::unix)
;=> {:parents {:user/osx #{:user/unix}},
     :ancestors {:user/osx #{:user/unix}},
     :descendants {:user/unix #{:user/osx}}}
```

Once you have a separate hierarchy in hand, you can provide it to `defmulti` to specify the derivation context, thus preserving the global hierarchy map.

9.2.6 Arbitrary dispatch for true maximum power

Until now, we've only exercised multimethods using a single privileged `:os` property, but this doesn't accentuate their true power. Instead, multimethods are fully open and can dispatch on the result of an arbitrary function, even one that can pull apart and/or combine its inputs into any form:

```
(defmulti  compile-cmd  (juxt :os compiler))

(defmethod compile-cmd [::osx "gcc"] [m]
  (str "/usr/bin/" (get m :c-compiler)))

(defmethod compile-cmd :default [m]
  (str "Unsure where to locate " (get m :c-compiler)))
```

The dispatch values for the new `compile-cmd` methods are vectors composed of the results of looking up the `:os` key and calling the `compiler` function defined earlier. You can now observe what happens when `compile-cmd` is called:

```
(compile-cmd osx)
;=> "/usr/bin/gcc"

(compile-cmd unix)
;=> "Unsure where to locate cc"
```

Using multimethods and the UDP is an interesting way to build abstractions. Multimethods and ad hoc hierarchies are open systems, allowing for polymorphic dispatch based on arbitrary functions. Clojure also provides a simpler model for creating abstractions and gaining the benefits of polymorphism—types, protocols, and records—which we'll cover next.

The handy-dandy juxt function

The `juxt` function is highly useful in defining multimethod dispatch functions. In a nutshell, `juxt` takes a bunch of functions and composes them into a function returning a vector of its argument(s) applied to each given function, as shown:

```
(def each-math (juxt + * - /))
(each-math 2 3)
;=> [5 6 -1 2/3]

((juxt take drop) 3 (range 9))
[(0 1 2) (3 4 5 6 7 8)]
```

Having a convenient and succinct way to build vectors of applied functions is powerful for defining understandable multimethods—although that's not the limit of `juxt`'s usefulness.

9.3 Types, protocols, and records

We showed in the previous section that Clojure multimethods provide a way to achieve runtime polymorphism based on arbitrary dispatch functions. Though extremely powerful, multimethods are sometimes less than ideal. Interposing a dispatch function into the polymorphism machinery isn't always conducive to raw speed. Likewise, dispatching on an arbitrary function is often overkill. Therefore, Clojure provides facilities for creating logically grouped polymorphic functions that are both simple and performant—types, records, and protocols. We'll delve into these topics in this section and introduce the concept of abstraction-oriented programming, predicated on the creation of logical groupings. But first, we'll discuss the simplest of the three topics, records, which you might recognize.

9.3.1 Records

Using maps as data objects is perfectly acceptable and has several lovely features. Chief among these is that maps require no declaration of any sort: you just use literal syntax to build them right on the spot. We showed this in section 7.2 when we built an object like this:

```
{:val 5, :l nil, :r nil}
```

This is handy but is missing things that are often desirable, the most significant of which is a type of its own. The object constructed here is some kind of map, but it isn't, as far as Clojure is concerned, a `TreeNode`. That means that when used in its simple form as we did here, there's no clean way[2] to determine whether any particular map is a `TreeNode` or not.

In such circumstances, records become a compelling[3] solution. You define a record type with a `defrecord` form. For example, a `defrecord` for `TreeNode` looks like this:

```
(defrecord TreeNode [val l r])
```

This creates a new Java class with a constructor that takes a value for each of the fields listed. It also imports that class into your current namespace so you can easily use it to create new instances.

Here's how to create an instance of the `TreeNode` record:

```
(TreeNode. 5 nil nil)
;=> #:user.TreeNode{:val 5, :l nil, :r nil}
```

---

[2] You could test a map for the existence of the keys `:val`, `:l`, and `:r`, a sort of duck-typing but on fields instead of methods. But because there exists a real possibility than some other kind of object may happen to have these keys but use them in a different way, undesirable complexity and/or unexpected behavior is likely. Fortunately, you can mitigate this risk by using namespace-qualified keywords. Despite the general agreement of experts that ducks are Kosher, we'd definitely classify this particular duck as unclean.

[3] There was a pre-Clojure 1.2 convention of attaching `:type` metadata to an object, which can be looked up with the type function, but this approach is rarely if ever needed moving forward.

Explicit importing of defrecord and deftype classes

It's important to note that when you define a `defrecord` and `deftype`, corresponding classes are generated. These classes are automatically imported into the same namespace where the `defrecord` and `deftype` declarations occur, but *not* in any other namespace. Instead, you *must explicitly import* `defrecord` and `deftype` classes using the `import` function or `:import` namespace declaration:

```
(ns my-cool-ns
  (:import joy.udp.TreeNode))
```

Loading a namespace via `:require` or `:use` won't be enough to import `defrecord` and `deftype` classes.

The use of `defrecord` buys you several important benefits. First of all, it provides a simple and specific idiom for documenting the expected fields of the object. But it also delivers several important performance improvements. A record will be created more quickly, consume less memory, and look up keys in itself more quickly than the equivalent array map or hash map. Data types can also store primitive values (byte, int, long, and so on), which take up considerably less memory than the equivalent boxed objects (Byte, Integer, Long, and so on) supported by other collection types.

That's a lot of benefit, so what does it cost you? The first cost we already mentioned—you must define the record type before using it. Another is that currently, records aren't printed in a way that the Clojure reader can read, unlike hash maps. This can be a problem if you're using Clojure's print functions to save off or transmit data. Here's what it looks like if we try, successfully, with a literal map and then again, unsuccessfully, with a record:

The downfall of defstructs

Clojure provides a `defstruct` mechanism, which can be viewed as a way to define a map that acts as an ad hoc class mechanism. These structs defined a set of keys that were required to exist in the map and could therefore not be removed via `dissoc`. With the advent of `defrecord`, the need for structs has been nearly eliminated, and therefore structs aren't covered in this book. But if you have a code base reliant on structs, then a record can replace them with minimal code changes, as highlighted here:

```
(defn change-age [p] (assoc p :age 286))

(defstruct person :fname :lname)
(change-age (struct person "Immanuel" "Kant"))
;=> {:fname "Immanuel", :lname "Kant", :age 286}

(defrecord Person [fname lname])
(change-age (Person. "Immanuel" "Kant"))
;=> #:user.Person{:fname "Immanuel", :lname "Kant", :age 286}
```

Note that the `change-age` function works with either structs or records—no change is required. Only the definition and the mechanism of instantiation need to be updated.

```
(read-string (pr-str {:val 5, :l nil, :r nil}))
;=> {:val 5, :l nil, :r nil}

(read-string (pr-str (TreeNode. 5 nil nil)))
; java.lang.RuntimeException: java.lang.Exception: No dispatch macro for:
```

This may change eventually, but there are some tricky problems yet to be worked out before records can be printed and read back in.

Other noteworthy differences between maps and records include

- Records, unlike maps, can't serve as functions.
- Records are never equal to maps with the same key/value mappings.

You still look up values in records by doing `(:keyword obj)`; it's just that if `obj` is a record, this code will run dramatically faster. By the way, that means destructuring will still work as well. Records support metadata using `with-meta` and `meta` just like other Clojure collections, and you can even redefine a record if desired to have different fields giving you the compiled performance of Java dynamically. All of these together mean you can build a lot of code on top of simple hash-map objects and then make minimal changes to switch to using records instead, gaining all the performance benefits we already covered.

You should understand records well enough to be able to reimplement the persistent binary tree from chapter 5 using `defrecord` instead of maps. This is shown in the following listing. Note that we had to add the `defrecord` and change the expressions in `xconj` where objects are created, but the `xseq` function is defined identically to how it was before.

Listing 9.2 Persistent binary tree built of records

```
(defrecord TreeNode [val l r])                                  <-- Define record type

(defn xconj [t v]                                               <-- Add to tree
  (cond
    (nil? t)       (TreeNode. v nil nil)
    (< v (:val t)) (TreeNode. (:val t) (xconj (:l t) v) (:r t))
    :else          (TreeNode. (:val t) (:l t) (xconj (:r t) v))))

(defn xseq [t]                                                  <-- Convert trees to seqs
  (when t
    (concat (xseq (:l t)) [(:val t)] (xseq (:r t)))))

(def sample-tree (reduce xconj nil [3 5 2 4 6]))                <-- Try it all out
(xseq sample-tree)
;=> (2 3 4 5 6)
```

You can `assoc` and `dissoc` any key you want—adding keys that weren't defined in the `defrecord` works, though they have the performance of a regular map. Perhaps more surprisingly, dissocing a key given in the record works but returns a regular map rather than a record. In this example, note that the return value is printed as a plain map, not with the `#:user.TreeNode` prefix of a record:

```
(dissoc (TreeNodePlus 5 nil nil) :l)
;=> {:val 5, :r nil}
```

A final benefit of records is how well they integrate with Clojure protocols. But to fully understand how they relate, we must first explore what protocols are.

9.3.2 Protocols

> *The establishment of protocols ... creates an obvious way for two people who are not directly communicating to structure independently developed code so that it works in a manner that remains coherent when such code is later combined.*
>
> —Kent M. Pitman (Pitman 2001)

A *protocol* in Clojure is simply a set of function signatures, each with at least one parameter, that are given a collective name. They fulfill a role somewhat like Java interfaces or C++ pure virtual classes—a class that claims to implement a particular protocol should provide specialized implementations of each of the functions in that protocol. Then, when any of those functions is called, the appropriate implementation is polymorphic on the type of the first parameter, just like Java. In fact, the first parameter to a protocol function corresponds to the target object (the thing to the left of the dot for a method call used in Java source) of a method in object-oriented parlance.

For example, consider what collections such as stacks (First In, Last Out: FILO) and queues (First In, First Out: FIFO) have in common. Each has a simple function for inserting a thing (call it `push`), a simple function for removing a thing (`pop`), and usually a function to see what would be removed if you removed a thing (`peek`). What we just gave you was an informal description of a protocol; all that's missing is the name. We can replace the changing third item of the acronym with an *X* and call objects that provide these functions FIXO. Note that besides stacks and queues, FIXO could include priority queues, pipes, and other critters.

So now let's look at that informal description rewritten as a formal Clojure definition:

```
(defprotocol FIXO
  (fixo-push [fixo value])
  (fixo-pop [fixo])
  (fixo-peek [fixo]))
```

...and that's it. The only reason we prefixed the function names with *fixo-* is so that they don't conflict with Clojure's built-in functions.[4] Besides that, it's hard to imagine how there could be much less ceremony, isn't it?

But in order for a protocol to do any good, something must implement it. Protocols are implemented using one of the *extend* forms: `extend`, `extend-type`,[5] or `extend-protocol`. Each of these does essentially the same thing, but `extend-type` and

[4] It would be better to fix this problem by defining FIXO in a new namespace and excluding from it the similarly named clojure.core functions, except this would be a distraction from the point of this section. We'll discuss interesting interactions between namespaces and protocols later in this chapter.

[5] Records are a specialized kind of data type, so `extend-type` is used for both. We'll look at data types later in this section.

`extend-protocol` are convenience macros for when you want to provide multiple functions for a given type. For example, the binary `TreeNode` from listing 9.2 is a record, so if we want to extend it, `extend-type` would be most convenient. Because `TreeNode` already has a function `xconj` that works just like `push` should, we'll start by implementing that:

```
(extend-type TreeNode
  FIXO
  (fixo-push [node value]
    (xconj node value)))

(xseq (fixo-push sample-tree 5/2))
;=> (2 5/2 3 4 5 6)
```

The first argument to `extend-type` is the class or interface that the entire rest of the form will be extending. Following the type name are one or more blocks, each starting with the name of the protocol to be extended and followed by one or more functions from that protocol to implement. So in the preceding example, we're implementing a single function `fixo-push` for `TreeNode` objects, and we call the existing `xconj` function. Got it? The reason this is better than simply defining a regular function named `fixo-push` is that protocols allow for polymorphism. That same function can have a different implementation for a different kind of object. Clojure vectors can act like stacks by extending `FIXO` to vectors:

```
(extend-type clojure.lang.IPersistentVector
  FIXO
  (fixo-push [vector value]
    (conj vector value)))

(fixo-push [2 3 4 5 6] 5/2)
;=> [2 3 4 5 6 5/2]
```

Here we're extending `FIXO` to an interface instead of a concrete class. This means that `fixo-push` is now defined for all classes that inherit from `IPersistentVector`. Note that we can now call `fixo-push` with either a vector or a `TreeNode`, and the appropriate function implementation is invoked.

Clojure-style mixins

As you proceed through this section, you'll notice that we extend the `FIXO` protocol's `fixo-push` function in isolation. This works fine for our purposes, but you might want to take note of the implications of this approach. Consider the following:

```
(use 'clojure.string)

(defprotocol StringOps (rev [s]) (upp [s]))

(extend-type String
  StringOps
  (rev [s] (clojure.string/reverse s)))

(rev "Works")
;=> "skroW"
```

(continued)

Defining the `StringOps` protocol and extending its `rev` function to `String` seems to work fine. But observe what happens when the protocol is again extended to cover the remaining `upp` function:

```
(extend-type String
  StringOps
  (upp [s] (clojure.string/upper-case s)))

(upp "Works")
;=> "WORKS"

(rev "Works?")
; IllegalArgumentException No implementation of method: :rev
;   of protocol: #'user/StringOps found for
;     class: java.lang.String
```

The reason for this exception is that for a protocol to be fully populated (all of its functions callable), it must be extended fully, per individual type. Protocol extension is at the granularity of the entire protocol and not at a per-function basis. This behavior seems antithetical to the common notion of a *mixin*—granules of discrete functionality that can be "mixed into" existing classes, modules, and so on. Clojure too has mixins, but it takes a slightly different approach:

```
(def rev-mixin {:rev clojure.string/reverse})

(def upp-mixin {:upp (fn [this] (.toUpperCase this))})

(def fully-mixed (merge upp-mixin rev-mixin))

(extend String StringOps fully-mixed)

(-> "Works" upp rev)
;=> SKROW
```

Mixins in Clojure refer to the creation of discrete maps containing protocol function implementations that are combined in such a way as to create a complete implementation of a protocol. Once mixed together (as in the Var `fully-mixed`), only then are types extended to protocols. As with many of Clojure's features, mixins and protocol extension are fully open.

What we've just done is impossible with Java interfaces or C++ classes, at least in the order we did it. With either of those languages, the concrete type (such as `TreeNode` or vector) must name *at the time it's defined* all the interfaces or classes it's going to implement. Here we went the other way around—both `TreeNode` and vectors were defined before the `FIXO` protocol even existed, and we easily extended `FIXO` to each of them. This matters in the real world because the concrete types and even the protocol could be provided by third-party libraries—possibly even different third-party libraries—and we could still match them up, provide implementations for the appropriate functions, and get back to work. All this without any adapters, wrappers, monkey-patching, or other incidental complexity getting in the way. In fact, *Clojure polymorphism lives in the protocol functions, not in the classes*, as shown in figure 9.7.

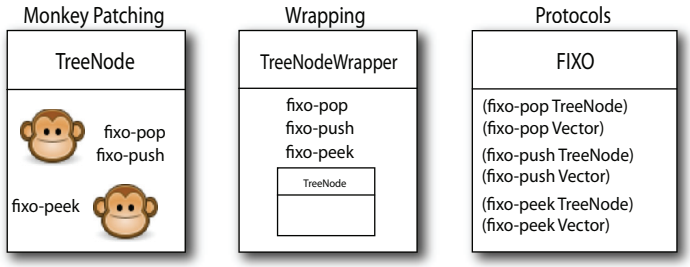

Figure 9.7 As opposed to the notion of monkey-patching and wrapping, the polymorphism in Clojure resides in the functions themselves and not in the classes worked with.

You can even extend a protocol to `nil` itself. You'd be forgiven for not immediately seeing why you'd want to do this; but consider how `TreeNode` implements `fixo-push`, and yet the `sample-tree` we're using was built using `xconj` instead. Trying to build up a tree the same way with `fixo-push` runs into a problem:

```
(reduce fixo-push nil [3 5 2 4 6 0])
; java.lang.IllegalArgumentException:
; No implementation of method: :fixo-push
;  of protocol: #'user/FIXO found for class: nil
```

The `xconj` implementation specifically handled the initial `nil` case, but because protocol methods dispatch on the first argument, we need special support from `extend` to get `fixo-push` to behave similarly. This is done by extending a protocol to the value `nil`, like this:

```
(extend-type nil
  FIXO
  (fixo-push [t v]
    (TreeNode. v nil nil)))

(xseq (reduce fixo-push nil [3 5 2 4 6 0]))
;=> (0 2 3 4 5 6)
```

All the options and arrangements of code allowed by `extend` can be disorienting, but one thing you can keep firmly in mind is that `extend` is always about a protocol. Each method listed in an `extend` form is implementing an intersection between a protocol and something else. That something else can be a concrete class, an interface, a record type, or even `nil`, but it's always being connected to a protocol.

See the following listing for complete implementations of `FIXO` for `TreeNode` and vectors. As mentioned in the sidebar, in order for the `FIXO` protocol to be fully realizable, each of its functions should be mixed in. But you might not always require that a protocol be fully realizable.

Listing 9.3 Complete implementations of `FIXO` for `TreeNode` and vector

```
(extend-type TreeNode
  FIXO
  (fixo-push [node value]
    (xconj node value))
  (fixo-peek [node]
    (if (:l node)
      (recur (:l node))
      (:val node)))
  (fixo-pop [node]
    (if (:l node)
      (TreeNode. (:val node) (fixo-pop (:l node)) (:r node))
      (:r node))))

(extend-type clojure.lang.IPersistentVector
  FIXO
  (fixo-push [vector value]
    (conj vector value))
  (fixo-peek [vector]
    (peek vector))
  (fixo-pop [vector]
    (pop vector)))
```

Delegate to xconj

Walk down left nodes to find smallest

Build new path down left to removed item

fixo-push is vector's conj

peek is peek

pop is pop

If you've done six impossible things this morning, why not round it off with breakfast at Milliways, the Restaurant at the End of the Universe?

—Douglas Adams

Each of the function bodies in the previous example have either had no code in common with each other, or called out to another function such as `xconj` for implementation details that they have in common. These techniques work well when there's a low level of commonality between the methods being implemented, but sometimes you have many methods of a protocol or even whole protocol implementations that you want to extend to multiple classes. In these cases, some languages would encourage you to create a base class that implements some or all of the methods and then inherit from that. Clojure has a different approach.

SHARING METHOD IMPLEMENTATIONS

Clojure doesn't encourage implementation inheritance, so although it's possible to inherit from concrete classes as needed for Java interoperability,[6] there's no way to use `extend` to provide a concrete implementation and then build another class on top of that. There are important reasons why Clojure intentionally avoids this, but regardless of the reasons, we're left with the question of how best to avoid repeating code when similar objects implement the same protocol method.

The simplest solution is to write a regular function that builds on the protocol's methods. For example, Clojure's own `into` takes a collection and uses the `conj` implementation provided by the collection. We can write a similar function for `FIXO` objects like this:

[6] Mechanisms that support something like Java-style implementation inheritance include `gen-class`, `proxy`, and extending protocol methods to Java abstract classes and interfaces.

```
(defn fixo-into [c1 c2]
  (reduce fixo-push c1 c2))

(xseq (fixo-into (TreeNode. 5 nil nil) [2 4 6 7]))
;=> (2 4 5 6 7)

(seq (fixo-into [5] [2 4 6 7]))
;=> (5 2 4 6 7)
```

But this is only an option when your function can be defined entirely in terms of the protocol's methods. If this isn't the case, you may need the more nuanced solution provided by the `extend` function. We mentioned it earlier but so far have only given examples of a macro built on top of it, `extend-type`. Though this and `extend-protocol` are frequently the most convenient way to implement protocol methods, they don't provide a natural way to mix in method implementations. The `extend` function takes a map for each protocol you want to implement, and you can build that map however you'd like, including by merging in implementations that are already defined. In the following listing, you should note how a `FIXO` implementation could be defined early using a map and extended to a protocol/record type later (while still maintaining every benefit of using the original map).

Listing 9.4 Using a map to extend `FIXO` to `TreeNode`

```
(def tree-node-fixo
  {:fixo-push (fn [node value]
                (xconj node value))
   :fixo-peek (fn [node]
                (if (:l node)
                  (recur (:l node))
                  (:val node)))
   :fixo-pop (fn [node]
               (if (:l node)
                 (TreeNode. (:val node) (fixo-pop (:l node)) (:r node))
                 (:r node)))})

(extend TreeNode FIXO tree-node-fixo)

(xseq (fixo-into (TreeNode. 5 nil nil) [2 4 6 7]))
;=> (2 4 5 6 7)
```

Define map of names to functions

Extend protocol using map

These record objects and the way protocols can be extended to them result in rather differently shaped code than the objects built out of closures that we showed in section 7.2. Often this ability to define the data and implementation separately is desirable, but you're likely to find yourself occasionally in a circumstance where closures may feel like a better fit than records, and yet you want to extend a protocol or interface, not just provide ad hoc method names as in section 7.2.

REIFY

The `reify` macro brings together all the power of function closures and all the performance and protocol participation of `extend` into a single form. For example, say you want a stack-like `FIXO` that's constrained to a certain fixed size. Any attempt to push items onto one of these fixed-fixos when it's already full will fail, and an unchanged

object will be returned. The wrinkle in the requirements that makes `reify` a reasonable option is that you'll want this size limit to be configurable. Thus you'll need a constructor or factory function, shown next, that takes the size limit and returns an object that will obey that limit.

Listing 9.5 Size-limited stack `FIXO` using `reify`

```
(defn fixed-fixo
  ([limit] (fixed-fixo limit []))
  ([limit vector]
    (reify FIXO
      (fixo-push [this value]
        (if (< (count vector) limit)
          (fixed-fixo limit (conj vector value))
          this))
      (fixo-peek [_]
        (peek vector))
      (fixo-pop [_]
        (pop vector)))))
```

Just like the `extend` forms, `reify` has method arglists that include the object itself. It's idiomatic to use name the argument `this` in methods where you need to use it and _ in methods where you ignore its value. But both these conventions should only be followed where natural.

NAMESPACED METHODS

A rough analogy can be drawn between protocols and Java interfaces.[7] We've noted some of the differences already, but it can be a useful analogy nonetheless. In such a comparison, where record types are concrete classes, you might see that Java packages and C++ namespaces are each like Clojure namespaces. It's normal in all three of these environments for the interface and the class to each be in a namespace, and not necessarily the same one. For example, probably few readers were surprised to see that when we made the class `IPersistentVector` extend the protocol `user/FIXO`, they were each from a different namespace or package.

One way this analogy breaks down is that methods of the protocol itself are namespaced in a way that Java and C++ interfaces aren't. In those languages, all methods of a class share the same effective namespace, regardless of interfaces they're implementing. In Clojure, the methods always use the same namespace as the protocol itself, which means a record or type can extend (via `extend`, `extend-type`, and so on) identically named methods of two different protocols without any ambiguity. This is a subtle feature, but it allows you to avoid a whole category of issues that can come up when trying to combine third-party libraries into a single codebase.

Note that because the methods share the namespace of their protocol, you can't have identically named methods in two different protocols if those protocols are in the same namespace. Because both are under the control of the same person, it's easy

[7] Those of you familiar with Haskell might recognize analogies to its typeclasses in our discussion.

to resolve this by moving one of the protocols to a different namespace or using more specific method names.

METHOD IMPLEMENTATIONS IN DEFRECORD

We've already shown how both protocols and interfaces can be extended to record types using the various `extend` forms, but there's another way to achieve similar results. Protocol and interface method implementations can be written directly inside a `defrecord` form, which ends up looking like the following.

Listing 9.6 Method implementations in `defrecord`

```
(defrecord TreeNode [val l r]
  FIXO
  (fixo-push [t v]
    (if (< v val)
      (TreeNode. val (fixo-push l v) r)
      (TreeNode. val l (fixo-push r v))))
  (fixo-peek [t]
    (if l
      (fixo-peek l)
      val))
  (fixo-pop [t]
    (if l
      (TreeNode. val (fixo-pop l) r)
      r)))

(def sample-tree2 (reduce fixo-push (TreeNode. 3 nil nil) [5 2 4 6]))
(xseq sample-tree2)
;=> (2 3 4 5 6)
```

Implement FIXO methods inline

Call method instead of using recur

This isn't only more convenient in many cases, but it can also produce dramatically faster code. Calling a protocol method like `fixo-peek` on a record type that implements it inline can be several times faster than calling the same method on an object that implements it via an `extend` form. Also note that the fields of the object are now available as locals—we use `val` instead of `(:val t)`.

Polymorphism and recur

Throughout this section, we've implemented the `fixo-peek` function using different methodologies, but a more subtle difference is worth noting. The first implementation uses `recur` for its recursive call as shown:

```
(fixo-peek [node]
  (if (:l node)
    (recur (:l node))
    (:val node)))
```

Because of the nature of `recur`, the first implementation of `fixo-peek` isn't polymorphic on the recursive call. But the second version of `fixo-peek` uses a different approach:

(continued)

```
(fixo-peek [t]
  (if l
    (fixo-peek l)
    val))
```

You'll notice that the recursive call in the second implementation is direct (mundane) and as a result is polymorphic. In the course of writing your own programs, this difference will probably not cause issues, but it's worth storing in the back of your mind.

Putting method definitions inside the `defrecord` form also allows you to implement Java interfaces and extend `java.lang.Object`, which isn't possible using any `extend` form. Because interface methods can accept and return primitive values as well as boxed objects, implementations of these in `defrecord` can also support primitives. This is important for interoperability and can provide ultimate performance parity with Java code.

We do need to note one detail of these inline method definitions in relation to `recur`. Specifically, uses of `recur` in these definitions can't provide a new target object: the initial argument will get the same value as the initial (non-`recur`) call to the method. For example, `fixo-push` takes args `t` and `v`, so if it used `recur`, only a single parameter would be given: the new value for the `v` arg.

9.3.3 Building from a more primitive base with deftype

You may have noticed we've been using our own function `xseq` throughout the examples in this section, instead of Clojure's `seq`. This shouldn't be necessary, as Clojure provides an `ISeqable` interface that its `seq` function can use—all we need to do is to have our own type implement `ISeqable`. But an attempt to do this with `defrecord` is doomed:

```
(defrecord InfiniteConstant [i]
  clojure.lang.ISeq
  (seq [this]
    (lazy-seq (cons i (seq this)))))
; java.lang.ClassFormatError: Duplicate method
;   name&signature in class file user/InfiniteConstant
```

This is because record types are maps and implement everything maps should—`seq` along with `assoc`, `dissoc`, `get`, and so forth. Because these are provided for us, we can't implement them again ourselves, and thus the preceding exception. For the rare case where you're building your own data structure instead of just creating application-level record types, Clojure provides a lower-level `deftype` construct that's similar to `defrecord` but doesn't implement anything at all, so implementing `seq` won't conflict with anything:

```
(deftype InfiniteConstant [i]
  clojure.lang.ISeq
  (seq [this]
```

```
    (lazy-seq (cons i (seq this)))))

(take 3 (InfiniteConstant. 5))
;=> (5 5 5)
```

But that also means that keyword lookups, `assoc`, `dissoc`, and so on will remain unimplemented unless we implement them ourselves:

```
(:i (InfiniteConstant. 5))
;=> nil
```

The fields we declared are still public and accessible (although you should try to avoid naming them the same at the methods in `java.lang.Object`); they just require normal Java interop forms to get at them:

```
(.i (InfiniteConstant. 5))
;=> 5
```

With all that in mind, the following listing is a final implementation of `TreeNode` using `deftype`, which lets us implement not only `ISeq` so that we can use `seq` instead of `xseq`, but also `IPersistentStack` so we can use `peek`, `pop`, and `conj` as well as the `fixo-` versions.

Listing 9.7 Implementing map interfaces with `deftype`

```
(deftype TreeNode [val l r]
  FIXO                                          <- Implement FIXO methods inline
  (fixo-push [_ v]
    (if (< v val)
      (TreeNode. val (fixo-push l v) r)
      (TreeNode. val l (fixo-push r v))))
  (fixo-peek [_]
    (if l
      (fixo-peek l)                             <- Call method instead of using recur
      val))
  (fixo-pop [_]
    (if l
      (TreeNode. val (fixo-pop l) r)
      r))

  clojure.lang.IPersistentStack                 <- Implement interfaces
  (cons [this v] (fixo-push this v))
  (peek [this] (fixo-peek this))
  (pop [this] (fixo-pop this))

  clojure.lang.Seqable
  (seq [t]
    (concat (seq l) [val] (seq r))))

(extend-type nil
  FIXO
  (fixo-push [t v]                              <- Redefine to use new TreeNode
    (TreeNode. v nil nil)))

(def sample-tree2 (into (TreeNode. 3 nil nil) [5 2 4 6]))
(seq sample-tree2)
;=> (2 3 4 5 6)
```

One final note about `deftype`—it's the one mechanism by which Clojure lets you create classes with volatile and mutable fields. We won't go into it here because using such classes is almost never the right solution. Only when you've learned how Clojure approaches identity and state, how to use reference types, what it means for a field to be volatile, and all the pitfalls related to that, should you even consider creating classes with mutable fields. By then, you'll have no problem understanding the official docs for `deftype`, and you won't need any help from us.

None of the examples we've shown in this section come close to the flexibility of multimethods. All protocol methods dispatch on just the type of the first argument. This is because that's what Java is good at doing quickly, and in many cases it's all the polymorphism that's needed. Clojure once again takes the practical route and makes the highest-performance mechanisms available via protocols, while providing more dynamic behavior than Java does and leaving multimethods on the table for when ultimate flexibility is required.

9.4 Putting it all together: a fluent builder for chess moves

People have been known to say that Java is a verbose programming language. This may be true when compared to the Lisp family of languages, but considerable mindshare has been devoted to devising ways to mitigate its verbosity. One popular technique is known as the *fluent builder* (Fowler 2005) and can be summed up as the chaining of Java methods to form a more readable and agile instance construction technique. In this section, we'll show a simple example of a fluent builder supporting the construction of chess move descriptions. We'll then explain how such a technique is unnecessary within Clojure and instead present an alternative approach that's simpler, concise, and more extensible. We'll leverage Clojure's records in the final solution, illustrating that Java's class-based paradigm is counter to Clojure's basic principles and often overkill for Java programs.

9.4.1 Java implementation

We'll start by identifying all of the component parts of a `Move` class including from and to squares, a flag indicating whether the move is a castling move, and also the desired promotion piece if applicable. In order to constrain the discussion, we'll limit our idea of a `Move` to those elements listed. The next step would be to create a simple class with its properties and a set of constructors, each taking some combination of the expected properties. We'd then generate a set of accessors for the properties, but not their corresponding mutators, because it's probably best for the move instances to be immutable.

Having created this simple class and rolled it out to the customers of the chess move API, we begin to notice that our users are sending into the constructor the `to` string before the `from` string, which is sometimes placed after the `promotion`, and so on. After some months of intense design and weeks of development and testing, we release the following elided chess move class:

```
public class FluentMove {
    String from, to, promotion = "";
    boolean castlep;

    public static MoveBuilder desc() { return new MoveBuilder(); }

    public String toString() {
        return "Move " + from +
           " to " + to +
           (castlep ? " castle" : "") +
           (promotion.length() != 0 ? " promote to " + promotion : "");
    }

    public static final class MoveBuilder {
        FluentMove move = new FluentMove();

        public MoveBuilder from(String from) {
            move.from = from; return this;
        }

        public MoveBuilder to(String to) {
            move.to = to; return this;
        }

        public MoveBuilder castle() {
            move.castlep = true; return this;
        }

        public MoveBuilder promoteTo(String promotion) {
            move.promotion = promotion; return this;
        }

        public FluentMove build() { return move; }
    }
}
```

For brevity's sake, our code has a lot of holes, such as missing checks for fence post errors, `null`, empty strings, assertions, and invariants; it does allow us to illustrate that the code provides a fluent builder given the following `main` method:

```
public static void main(String[] args) {
    FluentMove move = FluentMove.desc()
        .from("e2")
        .to("e4").build();

    System.out.println(move);

    move = FluentMove.desc()
        .from("a1")
        .to("c1")
        .castle().build();

    System.out.println(move);

    move = FluentMove.desc()
        .from("a7")
        .to("a8")
        .promoteTo("Q").build();

    System.out.println(move);
}
```

```
//  Move e2 to e4
//  Move a1 to c1 castle
//  Move a7 to a8 promote to Q
```

The original constructor ambiguities have disappeared, with the only trade-off being a slight increase in complexity of the implementation and the breaking of the common Java getter/setter idioms—both of which we're willing to live with. But if we'd started the chess move API as a Clojure project, the code would likely be a very different experience for the end user.

9.4.2 Clojure implementation

In lieu of Java's class-based approach, Clojure provides a core set of collection types, and as you might guess, its map type is a nice candidate for move representation:

```
{:from "e7", :to "e8", :castle? false, :promotion \Q}
```

Simple, no?

In a language like Java, it's common to represent everything as a class—to do otherwise is either inefficient, non-idiomatic, or outright taboo. Clojure prefers simplification, providing a set of composite types perfect for representing most categories of problems typically handled by class hierarchies. Using Clojure's composite types makes sense for one simple reason: existing functions, built on a sequence abstraction, *just work*:

```
(defn build-move [& pieces]
  (apply hash-map pieces))

(build-move :from "e7" :to "e8" :promotion \Q)

;=> {:from "e7", :to "e8", :promotion \Q}
```

In two lines, we've effectively replaced the Java implementation with an analogous, yet more flexible representation. The term *domain-specific language (DSL)* is often thrown around to describe code such as `build-move`, but to Clojure (and Lisps in general) the line between DSL and API is blurred. In the original `FluentMove` class, we required a cornucopia of code in order to ensure the API was agnostic of the ordering of move elements; using a map, we get that for free. Additionally, `FluentMove`, though relatively concise, was still bound by fundamental Java syntactical and semantic constraints.

There's one major problem with our implementation—it doesn't totally replace the Java solution. If you recall, the Java solution utilized the `toString` method to print its representative form. The existence of a polymorphic print facility in Java is nice, and it allows a class creator to define a default print representation for an object when sent to any Java print stream. This means that the same representation is used on the console, in log files, and so on. Using raw maps can't give us this same behavior, so how can we solve this problem?

USING RECORDS

If we instead use a record, then the solution is as simple as that shown next.

Listing 9.8 A chess move record

```
(defrecord Move [from to castle? promotion]
  Object
  (toString [this]
    (str "Move " (:from this)
         " to " (:to this)
         (if (:castle? this) " castle"
           (if-let [p (:promotion this)]
             (str " promote to " p)
             "")))))
```

As we mentioned in the previous section, within the body of a record we can take up to two actions: participate in a protocol, or override any of the methods in the `java.lang.Object` class. For the `Move` record, we override `toString` in order to allow it to participate in Java's overarching polymorphic print facility, as shown:

```
(str (Move. "e2" "e4" nil nil))
;=> "Move e2 to e4"

(.println System/out (Move. "e7" "e8" nil \Q))
; Move e7 to e8 promote to Q
```

We've once again gone back to positional construction using records, but as we'll show, Clojure even has an answer for this.

SEPARATION OF CONCERNS

Both `FluentMove` and `build-move` make enormous assumptions about the form of the data supplied to them and do no validation of the input. For `FluentMove`, object-oriented principles dictate that the validation of a well-formed move (not a legal move, mind you) should be determined by the class itself. There are a number of problems with this approach, the most obvious being that to determine whether a move is well-formed, the class needs information about the rules of chess. We can rewrite `FluentMove` to throw an exception to prevent illegal moves from being constructed, but the root problem still remains—`FluentMove` instances are too smart. Perhaps you don't see this as a problem, but if we were to extend our API to include other aspects of the game of chess, then we'll find that bits of overlapping chess knowledge would be scattered throughout the class hierarchy. By viewing the move structure as a value, Clojure code provides some freedom in the implementation of a total solution, as shown:

```
(defn build-move [& {:keys [from to castle? promotion]}]
  {:pre [from to]}
  (Move. from to castle? promotion))

(str (build-move :from "e2" :to "e4"))
;=> "Move e2 to e4"
```

By wrapping the `Move` constructor in a `build-move` function, we put the smarts of constructing moves there instead of in the type itself. In addition, using a precondition, we specified the required fields, and by using Clojure's named parameters and argument destructuring we've again ensured argument order independence. As a final added advantage, Clojure's records are maps and as a result can operate in almost every circumstance where a map would. As author Rich Hickey proclaimed, any new class in general is itself an island, unusable by *any* existing code written by anyone, anywhere. So our point is this: consider throwing the baby out with the bath water.

9.5 Summary

Clojure disavows the typical object-oriented model of development. But that's not to say that it completely dismisses all that OOP stands for. Instead, Clojure wholeheartedly touts the virtues of interface-oriented programming (or abstraction-oriented programming, as we've called it), in addition to runtime polymorphism. But in both cases, the way that Clojure presents these familiar topics is quite different from what you might be accustomed to. In almost every circumstance, Clojure's abstraction-oriented facilities will sufficiently represent your problem domain, but there may be times when they simply can't. We'll preach the virtues of abstractions more throughout the rest of the book, but for now we're compelled to take a side path into an explorations of Java interoperability.

FUNCTIONAL PROGRAMMING

THE Joy OF Clojure

Fogus • Houser

If you've seen how dozens of lines of Java or Ruby can dissolve into just a few lines of Clojure, you'll know why the authors of this book call it a "joyful language." Clojure is a dialect of Lisp that runs on the JVM. It combines the nice features of a scripting language with the powerful features of a production environment—features like persistent data structures and clean multithreading that you'll need for industrial-strength application development.

The Joy of Clojure goes beyond just syntax to show you how to write fluent and idiomatic Clojure code. You'll learn a functional approach to programming and will master Lisp techniques that make Clojure so elegant and efficient. The book gives you easy access to hard software areas like concurrency, interoperability, and performance. And it shows you how great it can be to think about problems the Clojure way.

What's Inside

- The *what* and *why* of Clojure
- How to work with macros
- How to do elegant application design
- Functional programming idioms

Written for programmers coming to Clojure from another programming background—no prior experience with Clojure or Lisp is required.

Michael Fogus is a member of Clojure/core with experience in distributed simulation, machine vision, and expert systems. **Chris Houser** is a key contributor to Clojure who has implemented several of its features.

For online access to the authors and a free ebook for owners of this book, go to manning.com/TheJoyofClojure

"You'll learn fast!"
—From the foreword by Steve Yegge, Google

"Simply unputdownable!"
—Baishampayan Ghose (BG) Qotd, Inc.

"Discover the *why*, not just the *how* of Clojure."
—Federico Tomassetti Politecnico di Torino

"What Irma Rombauer did for cooking, Fogus and Houser have done for Clojure."
—Phil Hagelberg, Sonian

ISBN 13: 978-1-935182-64-1
ISBN 10: 1-935182-64-1
54499
9 781935 182641

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]