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 1

brief contents

1 Clojure philosophy

This chapter covers

- The Clojure way
- Why a(nother) Lisp?
- Functional programming
- Why Clojure isn't especially object-oriented

Learning a new language generally requires significant investment of thought and effort, and it is only fair that programmers expect each language they consider learning to justify that investment. Clojure was born out of creator Rich Hickey's desire to avoid many of the complications, both inherent and incidental, of managing state using traditional object-oriented techniques. Thanks to a thoughtful design based in rigorous programming language research, coupled with a fervent look toward practicality, Clojure has blossomed into an important programming language playing an undeniably important role in the current state of the art in language design. On one side of the equation, Clojure utilizes Software Transactional Memory (STM), agents, a clear distinction between identity and value types, arbitrary polymorphism, and functional programming to provide an environment conducive to making sense of state in general, and especially in the face of concurrency. On the other side, Clojure shares a symbiotic relationship with the

Java Virtual Machine, thus allowing prospective developers to avoid the costs of maintaining yet another infrastructure while leveraging existing libraries.

In the grand timeline of programming language history, Clojure is an infant; but its colloquialisms (loosely translated as "best practices" or idioms) are rooted[1] in 50 years of Lisp, as well as 15 years of Java history. Additionally, the enthusiastic community that has exploded since its introduction has cultivated its own set of unique idioms. As mentioned in the preface, the idioms of a language help to define succinct representations of more complicated expressions. Although we will certainly cover idiomatic Clojure code, we will also expand into deeper discussions of the "why" of the language itself.

In this chapter, we'll discuss the weaknesses in existing languages that Clojure was designed to address, how it provides strength in those areas, and many of the design decisions Clojure embodies. We'll also look at some of the ways existing languages have influenced Clojure, and define terms that will be used throughout the book.

1.1 The Clojure way

We'll start slowly.

Clojure is an opinionated language—it doesn't try to cover all paradigms or provide every checklist bullet-point feature. Instead it provides the features needed to solve all kinds of real-world problems the Clojure way. To reap the most benefit from Clojure, you'll want to write your code with the same vision as the language itself. As we walk through the language features in the rest of the book, we'll mention not just what a feature does, but why it's there and how best to take advantage of it.

But before we get to that, we'll first take a high-level view of some of Clojure's most important philosophical underpinnings. Figure 1.1 lists some broad goals that Rich Hickey had in mind while designing Clojure and some of the more specific decisions that are built into the language to support these goals.

As the figure illustrates, Clojure's broad goals are formed from a confluence of supporting goals and functionality, which we will touch on in the following subsections.

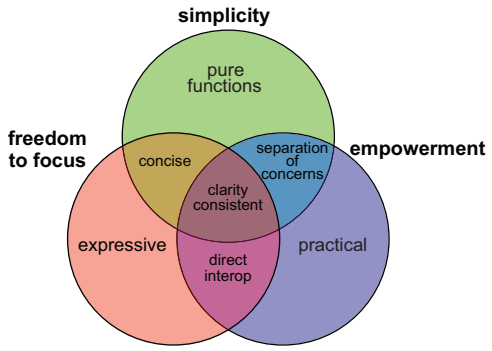

Figure 1.1 Broad goals of Clojure: this figure shows some of the concepts that underlie the Clojure philosophy, and how they intersect.

1.1.1 Simplicity

It's hard to write simple solutions to complex problems. But every experienced programmer has also stumbled on areas where we've made things more complex than necessary, what you might call

[1] While drawing on the traditions of Lisps (in general) and Java, Clojure in many ways stands as a direct challenge to them for change.

incidental complexity as opposed to complexity that's *essential* to the task at hand (Moseley 2006). Clojure strives to let you tackle complex problems involving a wide variety of data requirements, multiple concurrent threads, independently developed libraries, and so on without adding incidental complexity. It also provides tools reducing what at first glance may seem like essential complexity. The resulting set of features may not always seem simple, especially when they're still unfamiliar, but as you read through this book we think you'll come to see how much complexity Clojure helps strip away.

One example of incidental complexity is the tendency of modern object-oriented languages to require that every piece of runnable code be packaged in layers of class definitions, inheritance, and type declarations. Clojure cuts through all this by championing the *pure function*, which takes a few arguments and produces a return value based solely on those arguments. An enormous amount of Clojure is built from such functions, and most applications can be too, which means that there's less to think about when trying to solve the problem at hand.

1.1.2 Freedom to focus

Writing code is often a constant struggle against distraction, and every time a language requires you to think about syntax, operator precedence, or inheritance hierarchies, it exacerbates the problem. Clojure tries to stay out of your way by keeping things as simple as possible, not requiring you to go through a compile-and-run cycle to explore an idea, not requiring type declarations, and so on. It also gives you tools to mold the language itself so that the vocabulary and grammar available to you fit as well as possible to your problem domain—Clojure is *expressive*. It packs a punch, allowing you to perform highly complicated tasks succinctly without sacrificing comprehensibility.

One key to delivering this freedom is a commitment to dynamic systems. Almost everything defined in a Clojure program can be redefined, even while the program is running: functions, multimethods, types, type hierarchies, and even Java method implementations. Though redefining things on the fly might be scary on a production system, it opens a world of amazing possibilities in how you think about writing programs. It allows for more experimentation and exploration of unfamiliar APIs, and it adds an element of fun that can sometimes be impeded by more static languages and long compilation cycles.

But Clojure's not just about having fun. The fun is a by-product of giving programmers the power to be more productive than they ever thought imaginable.

1.1.3 Empowerment

Some programming languages have been created primarily to demonstrate some nugget of academia or to explore certain theories of computation. Clojure is *not* one of these. Rich Hickey has said on numerous occasions that Clojure has value to the degree that it lets you build interesting and useful applications.

To serve this goal, Clojure strives to be practical—a tool for getting the job done. If a decision about some design point in Clojure had to weigh the trade-offs between the practical solution and a clever, fancy, or theoretically pure solution, usually the practical solution won out. Clojure could try to shield you from Java by inserting a comprehensive API between the programmer and the libraries, but this could make the use of third-party Java libraries more clumsy. So Clojure went the other way: direct, wrapper-free, compiles-to-the-same-bytecode access to Java classes and methods. Clojure strings are Java strings; Clojure function calls are Java method calls—it's simple, direct, and practical.

The decision to use the Java Virtual Machine (JVM) itself is a clear example of this practicality. The JVM has some technical weaknesses such as startup time, memory usage, and lack of *tail-call optimization*[2] (TCO). But it's also an amazingly practical platform—it's mature, fast, and widely deployed. It supports a variety of hardware and operating systems and has a staggering number of libraries and support tools available, all of which Clojure can take advantage of because of this supremely practical decision.

With direct method calls, `proxy`, `gen-class`, `gen-interface` (see chapter 10), `reify`, `definterface`, `deftype`, and `defrecord` (see section 9.3), Clojure works hard to provide a bevy of interoperability options, all in the name of helping you get your job done. Practicality is important to Clojure, but many other languages are practical as well. You'll start to see some ways that Clojure really sets itself apart by looking at how it avoids muddles.

1.1.4 Clarity

> *When beetles battle beetles in a puddle paddle battle and the beetle battle puddle is a puddle in a bottle they call this a tweetle beetle bottle puddle paddle battle muddle.*
>
> —Dr. Seuss

Consider what might be described as a simple snippet of code in a language like Python:

```
x = [5]
process(x)
x[0] = x[0] + 1
```

After executing this code, what's the value of `x`? If you assume `process` doesn't change the contents of `x` at all, it should be `[6]`, right? But how can you make that assumption? Without knowing exactly what `process` does, and whatever function it calls does, and so on, you can't be sure at all.

Even if you're sure `process` doesn't change the contents of `x`, add multithreading and now you have another whole set of concerns. What if some other thread changes

[2] Don't worry if you don't know what tail-call optimization is. Also don't worry if you *do* know what TCO is and think the JVM's lack of it is a critical flaw for a Lisp or functional language such as Clojure. All your concerns will be addressed in section 7.3. Until then, just relax.

x between the first and third lines? Worse yet, what if something is setting x at the moment the third line is doing its assignment—are you sure your platform guarantees an atomic write to that variable, or is it possible that the value will be a corrupted mix of multiple writes? We could continue this thought exercise in hopes of gaining some clarity, but the end result would be the same—what you have ends up not being clear at all, but the opposite: a muddle.

Clojure strives for code clarity by providing tools to ward off several different kinds of muddles. For the one just described, it provides immutable locals and persistent collections, which together eliminate most of the single- and multithreaded issues all at once.

You can find yourself in several other kinds of muddles when the language you're using merges unrelated behavior into a single construct. Clojure fights this by being vigilant about separation of concerns. When things start off separated, it clarifies your thinking and allows you to recombine them only when and to the extent that doing so is useful for a particular problem. Table 1.1 contrasts common approaches that merge concepts together in some other languages with separations of similar concepts in Clojure that will be explained in greater detail throughout this book.

Table 1.1 Separation of concerns in Clojure

Conflated	Separated	Where
Object with mutable fields	Values *from* identities	Chapter 4 and section 5.1
Class acts as namespace for methods	Function namespaces *from* type namespaces	Sections 8.2 and 8.3
Inheritance hierarchy made of classes	Hierarchy of names *from* data and functions	Chapter 8
Data and methods bound together lexically	Data objects *from* functions	Sections 6.1 and 6.2 and chapter 8
Method implementations embedded throughout class inheritance chain	Interface declarations *from* function implementations	Sections 8.2 and 8.3

It can be hard at times to tease apart these concepts in our own minds, but accomplishing it can bring remarkable clarity and a sense of power and flexibility that's worth the effort. With all these different concepts at your disposal, it's important that the code and data you work with express this variety in a consistent way.

1.1.5 Consistency

Clojure works to provide consistency in two specific ways: consistency of syntax and of data structures.

Consistency of syntax is about the similarity in form between related concepts. One simple but powerful example of this is the shared syntax of the `for` and `doseq` macros.

They don't do the same thing—`for` returns a lazy seq whereas `doseq` is for generating side effects—but both support the same mini-language of nested iteration, destructuring, and `:when` and `:while` guards. The similarities stand out when comparing the following examples:

```
(for [x [:a :b], y (range 5) :when (odd? y)] [x y])
;=> ([:a 1] [:a 3] [:b 1] [:b 3])

(doseq [x [:a :b], y (range 5) :when (odd? y)] (prn x y))
; :a 1
; :a 3
; :b 1
; :b 3
;=> nil
```

The value of this similarity is having to learn only one basic syntax for both situations, as well as the ease with which you can convert any particular usage of one form to the other if that becomes necessary.

Likewise, the consistency of data structures is the deliberate design of all of Clojure's persistent collection types to provide interfaces as similar to each other as possible, as well as to make them as broadly useful as possible. This is actually an extension of the classic Lisp "code is data" philosophy. Clojure data structures aren't used just for holding large amounts of application data, but also to hold the expression elements of the application itself. They're used to describe destructuring forms and to provide named options to various built-in functions. Where other object-oriented languages might encourage applications to define multiple incompatible classes to hold different kinds of application data, Clojure encourages the use of compatible map-like objects.

The benefit of this is that the same set of functions designed to work with Clojure data structures can be applied to all these contexts: large data stores, application code, and application data objects. You can use `into` to build any of these types, `seq` to get a lazy seq to walk through them, `filter` to select elements of any of them that satisfy a particular predicate, and so on. Once you've grown accustomed to having the richness of all these functions available everywhere, dealing with a Java or C++ application's `Person` or `Address` class will feel constraining.

Simplicity, freedom to focus, empowerment, consistency, and clarity.

Nearly every element of the Clojure programming language is designed to promote these goals. When writing Clojure code, if you keep in mind the desire to maximize simplicity, empowerment, and the freedom to focus on the real problem at hand, we think you'll find Clojure provides you the tools you need to succeed.

1.2 Why a(nother) Lisp?

By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems.

—Alfred North Whitehead

Go to any open source project hosting site and perform a search for the term "Lisp interpreter." You'll likely get a cyclopean mountain[3] of results from this seemingly innocuous term. The fact of the matter is that the history of computer science is littered (Fogus 2009) with the abandoned husks of Lisp implementations. Well-intentioned Lisps have come and gone and been ridiculed along the way, and still tomorrow the search results will have grown almost without bounds. Bearing in mind this legacy of brutality, why would anyone want to base their brand-new programming language on the Lisp model?

1.2.1 Beauty

Lisp has attracted some of the brightest minds in the history of computer science. But an argument from authority is insufficient, so you shouldn't judge Lisp on this alone. The real value in the Lisp family of languages can be directly observed through the activity of using it to write applications. The Lisp style is one of expressivity and empowerment, and in many cases outright beauty. Joy awaits the Lisp neophyte. The original Lisp language as defined by John McCarthy in his earth-shattering essay "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I" (McCarthy 1960) defined the whole language in terms of only seven functions and two special forms: `atom`, `car`, `cdr`, `cond`, `cons`, `eq`, `quote`, `lambda`, and `label`.

Through the composition of those nine forms, McCarthy was able to describe the whole of computation in a way that takes your breath away. Computer programmers are perpetually in search of beauty, and more often than not, this beauty presents itself in the form of simplicity. Seven functions and two special forms. It doesn't get more beautiful than that.

1.2.2 Extreme flexibility

Why has Lisp persevered for more than 50 years while countless other languages have come and gone? There are probably complex reasons, but chief among them is likely the fact that Lisp as a language genotype (Tarver 2008) fosters language flexibility in the extreme. Newcomers to Lisp are sometimes unnerved by its pervasive use of parentheses and prefix notation, which is different than non-Lisp programming languages. The regularity of this behavior not only reduces the number of syntax rules you have to remember, but also makes the writing of macros trivial. We'll look at macros in more detail in chapter 8, but to whet your appetite we'll take a brief look at one now. It's an example that we'll get working on in a moment:

```
(defn query [max]
  (SELECT [a b c]
    (FROM X
      (LEFT-JOIN Y :ON (= X.a Y.b)))
    (WHERE (AND (< a 5) (< b ~max)))))
```

[3] ...of madness.

We hope some of those words look familiar to you, because this isn't a book on SQL. Regardless, our point here is that Clojure doesn't have SQL support built in. The words `SELECT`, `FROM`, and so forth aren't built-in forms. They're also not regular functions, because if `SELECT` were, then the use of `a`, `b`, and `c` would be an error, because they haven't been defined yet.

So what does it take to define a domain-specific language (DSL) like this in Clojure? Well, it's not production-ready code and doesn't tie into any real database servers; but with just one macro and the three functions shown in listing 1.1, the preceding query returns these handy values:

```
(query 5)
;=> ["SELECT a, b, c FROM X LEFT JOIN Y ON (X.a = Y.b)
      WHERE ((a < 5) AND (b < ?))"
     [5]]
```

Note that some words such as `FROM` and `ON` are taken directly from the input expression, whereas others such as `~max` and `AND` are treated specially. The `max` that was given the value `5` when the query was called is extracted from the literal SQL string and provided in a separate vector, perfect for using in a prepared query in a way that will guard against SQL-injection attacks. The `AND` form was converted from the prefix notation of Clojure to the infix notation required by SQL.

Listing 1.1 A domain-specific language for embedding SQL queries in Clojure

```
(ns joy.sql
  (:use [clojure.string :as str :only []]))

(defn expand-expr [expr]
  (if (coll? expr)
    (if (= (first expr) `unquote)
      "?"
      (let [[op & args] expr]
        (str "(" (str/join (str " " op " ")
                           (map expand-expr args)) ")")))
    expr))

(declare expand-clause)

(def clause-map
  {'SELECT    (fn [fields & clauses]
                (apply str "SELECT " (str/join ", " fields)
                       (map expand-clause clauses)))
   'FROM      (fn [table & joins]
                (apply str " FROM " table
                       (map expand-clause joins)))
   'LEFT-JOIN (fn [table on expr]
                (str " LEFT JOIN " table
                     " ON " (expand-expr expr)))
   'WHERE     (fn [expr]
                (str " WHERE " (expand-expr expr)))})

(defn expand-clause [[op & args]]
  (apply (clause-map op) args))
```

Use core string functions

Handle unsafe literals

Convert prefix to infix

Support each kind of clause

Call appropriate converter

```
(defmacro SELECT [& args]
  [(expand-clause (cons 'SELECT args))
   (vec (for [n (tree-seq coll? seq args)
              :when (and (coll? n) (= (first n) `unquote))]
          (second n)))])
```

Provide main entrypoint macro

But the point here isn't that this is a particularly good SQL DSL—more complete ones are available.[4] Our point is that once you have the skill to easily create a DSL like this, you'll recognize opportunities to define your own that solve much narrower, application-specific problems than SQL does. Whether it's a query language for an unusual non-SQL datastore, a way to express functions in some obscure math discipline, or some other application we as authors can't imagine, having the flexibility to extend the base language like this, without losing access to any of the language's own features, is a game-changer.

Although we shouldn't get into too much detail about the implementation, take a brief look at listing 1.1 and follow along as we discuss important aspects of its implementation.

Reading from the bottom up, you'll notice the main entry point, the `SELECT` macro. This returns a vector of two items—the first is generated by calling `expand-clause`, which returns the converted query string, whereas the second is another vector of expressions marked by ~ in the input. The ~ is known as *unquote* and we discuss its more common uses in chapter 8. Also note the use of `tree-seq` here to succinctly extract items of interest from a tree of values, namely the input expression.

The `expand-clause` function takes the first word of a clause, looks it up in the `clause-map`, and calls the appropriate function to do the actual conversion from Clojure s-expression to SQL string. The `clause-map` provides the specific functionality needed for each part of the SQL expression: inserting commas or other SQL syntax, and sometimes recursively calling `expand-clause` when subclauses need to be converted. One of these is the `WHERE` clause, which handles the general conversion of prefix expressions to the infix form required by SQL by delegating to the `expand-expr` function.

Overall, the flexibility of Clojure demonstrated in this example comes largely from the fact that macros accept code forms, such as the SQL DSL example we showed, and can treat them as data—walking trees, converting values, and more. This works not only because code can be treated as data, but because in a Clojure program, code *is* data.

1.2.3 Code is data

The notion of "code is data" is difficult to grasp at first. Implementing a programming language where code shares the same footing as its comprising data structures presupposes a fundamental malleability of the language itself. When your language is represented as the inherent data structures, the language itself can manipulate its own

[4] One of note is ClojureQL at http://gitorious.org/clojureql.

structure and behavior (Graham 1995). You may have visions of Ouroboros after reading the previous sentence, and that wouldn't be inappropriate, because Lisp can be likened to a self-licking lollypop—more formally defined as *homoiconicity*. Lisp's homoiconicity takes a great conceptual leap in order to fully grasp, but we'll lead you toward that understanding throughout this book in hopes that you too will come to realize the inherent power.

There's a joy in learning Lisp for the first time, and if that's your experience coming into this book then we welcome you—and envy you.

1.3 Functional programming

Quick, what does *functional programming* mean? Wrong answer.

Don't be too discouraged, however—we don't really know the answer either. Functional programming is one of those computing terms[5] that has a nebulous definition. If you ask 100 programmers for their definition, you'll likely receive 100 different answers. Sure, some definitions will be similar, but like snowflakes, no two will be exactly the same. To further muddy the waters, the cognoscenti of computer science will often contradict one another in their own independent definitions. Likewise, the basic structure of any definition of functional programming will be different depending on whether your answer comes from someone who favors writing their programs in Haskell, ML, Factor, Unlambda, Ruby, or Qi. How can *any* person, book, or language claim authority for functional programming? As it turns out, just as the multitudes of unique snowflakes are all made mostly of water, the core of functional programming across all meanings has its core tenets.

1.3.1 A workable definition of functional programming

Whether your own definition of functional programming hinges on the lambda calculus, monadic I/O, delegates, or `java.lang.Runnable`, your basic unit of currency is likely to be some form of procedure, function, or method—herein lies the root. Functional programming concerns and facilitates the application and composition of functions. Further, for a language to be considered functional, its notion of function must be *first-class*. The functions of a language must be able to be stored, passed, and returned just like any other piece of data within that language. It's beyond this core concept that the definitions branch toward infinity, but thankfully, it's enough to start. Of course, we'll also present a further definition of Clojure's style of functional programming that includes such topics as purity, immutability, recursion, laziness, and referential transparency, but those will come later in chapter 7.

1.3.2 The implications of functional programming

Object-oriented programmers and functional programmers will often see and solve a problem in different ways. Whereas an object-oriented mindset will foster the

[5] Quick, what's the definition of combinator? How about cloud computing? Enterprise? SOA? Web 2.0? Real-world? Hacker? Often it seems that the only term with a definitive meaning is "yak shaving."

approach of defining an application domain as a set of nouns (classes), the functional mind will see the solution as the composition or verbs (functions). Though both programmers may in all likelihood generate equivalent results, the functional solution will be more succinct, understandable, and reusable. Grand claims indeed! We hope that by the end of this book you'll agree that functional programming fosters elegance in programming. It takes a shift in mindset to start from thinking in nouns to arrive at thinking in verbs, but the journey will be worthwhile. In any case, we think there's much that you can take from Clojure to apply to your chosen language—if only you approach the subject with an open mind.

1.4 Why Clojure isn't especially object-oriented

Elegance and familiarity are orthogonal.

—Rich Hickey

Clojure was born out of frustration provoked in large part by the complexities of concurrent programming, complicated by the weaknesses of object-oriented programming in facilitating it. This section explores these weaknesses and lays the groundwork for why Clojure is functional and not object-oriented.

1.4.1 Defining terms

Before we begin, it's useful to define terms.[6]

The first important term to define is *time*. Simply put, time refers to the relative moments when events occur. Over time, the properties associated with an entity—both static and changing, singular or composite—will form a concrescence (Whitehead 1929) and be logically deemed its *identity*. It follows from this that at any given time, a snapshot can be taken of an entity's properties defining its *state*. This notion of state is an immutable one because it's not defined as a mutation in the entity itself, but only as a manifestation of its properties at a given moment in time. Imagine a child's flip book, as seen in figure 1.2, to understand the terms fully.

It's important to note that in the canon of object-oriented programming, there's no clear distinction between state and identity. In other words, these two ideas are

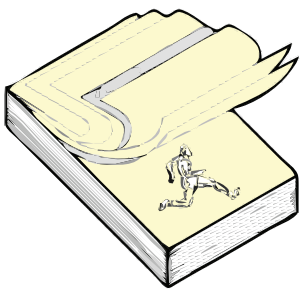

Figure 1.2 The Runner: a child's flip book serves to illustrate Clojure's notions of state, time, and identity. The book itself represents the identity. Whenever you wish to show a change in the illustration, you draw another picture and add it to the end of your flip book. The act of flipping the pages therefore represents the states over time of the image within. Stopping at any given page and observing the particular picture represents the state of the Runner at that moment in time.

[6] These terms are also defined and elaborated on in Rich Hickey's presentation, "Are We There Yet?" (Hickey 2009).

Figure 1.3 The Mutable Runner: modeling state change with mutation requires that you stock up on erasers. Your book becomes a single page, requiring that in order to model changes, you must physically erase and redraw the parts of the picture requiring change. Using this model, you should see that mutation destroys all notion of time, and state and identity become one.

conflated into what's commonly referred to as *mutable state*. The classical object-oriented model allows unrestrained mutation of object properties without a willingness to preserve historical states. Clojure's implementation attempts to draw a clear separation between an object's state and identity as they relate to time. To state the difference to Clojure's model in terms of the aforementioned flip book, the mutable state model is different, as seen in figure 1.3.

Immutability lies at the cornerstone of Clojure, and much of the implementation ensures that immutability is supported efficiently. By focusing on immutability, Clojure eliminates entirely the notion of *mutable state* (which is an oxymoron) and instead expounds that most of what's meant by objects are instead values. *Value* by definition refers to an object's constant representative[7] amount, magnitude, or epoch. You might ask yourself: what are the implications of the value-based programming semantics of Clojure?

Naturally, by adhering to a strict model of immutability, concurrency suddenly becomes a simpler (although not simple) problem, meaning if you have no fear that an object's state will change, then you can promiscuously share it without fear of concurrent modification. Clojure instead isolates value change to its reference types, as we'll show in chapter 11. Clojure's reference types provide a level of indirection to an identity that can be used to obtain consistent, if not always current, states.

1.4.2 Imperative "baked in"

Imperative programming is the dominant programming paradigm today. The most unadulterated definition of an imperative programming language is one where a sequence of statements mutates program state. During the writing of this book (and likely for some time beyond), the preferred flavor of imperative programming is the object-oriented style. This fact isn't inherently bad, because there are countless successful software projects built using object-oriented imperative programming techniques. But from the context of concurrent programming, the object-oriented imperative model is self-cannibalizing. By allowing (and even promoting) unrestrained mutation via *variables*, the imperative model doesn't directly support concurrency. Instead, by allowing a maenadic approach to mutation, there are no guarantees that any variable contains the expected value. Object-oriented programming takes this one step further by aggregating state in object internals. Though individual methods may be thread-safe through locking schemes, there's no way to ensure a consistent

[7] Some entities have no representative value—Pi is an example. But in the realm of computing, where we're ultimately referring to finite things, this is a moot point.

object state across multiple method calls without expanding the scope of potentially complex locking scheme(s). Clojure instead focuses on functional programming, immutability, and the distinction between state, time, and identity. But object-oriented programming isn't a lost cause. In fact, there are many aspects that are conducive to powerful programming practice.

1.4.3 Most of what OOP gives you, Clojure provides

It should be made clear that we're not attempting to mark object-oriented programmers as pariahs. Instead, it's important that we identify the shortcomings of object-oriented programming (OOP) if we're ever to improve our craft. In the next few subsections we'll also touch on the powerful aspects of OOP and how they're adopted, and in some cases improved, by Clojure.

POLYMORPHISM AND THE EXPRESSION PROBLEM

Polymorphism is the ability of a function or method to have different definitions depending on the type of the target object. Clojure provides polymorphism via both multimethods and protocols, and both mechanisms are more open and extensible than polymorphism in many languages.

Listing 1.2 Clojure's polymorphic protocols

```
(defprotocol Concatenatable
  (cat [this other]))

(extend-type String
  Concatenatable
  (cat [this other]
    (.concat this other)))

(cat "House" " of Leaves")
;=> "House of Leaves"
```

What we've done in listing 1.2 is to define a *protocol* named `Concatenatable` that groups one or more functions (in this case only one, `cat`) that define the set of functions provided. That means the function `cat` will work for any object that fully satisfies the protocol `Concatenatable`. We then *extend* this protocol to the `String` class and define the specific implementation—a function body that concatenates the argument `other` onto the string `this`. We can also extend this protocol to another type:

```
(extend-type java.util.List
  Concatenatable
  (cat [this other]
    (concat this other)))

(cat [1 2 3] [4 5 6])
;=> (1 2 3 4 5 6)
```

So now the protocol has been extended to two different types, `String` and `java.util.List`, and thus the `cat` function can be called with either type as its first argument—the appropriate implementation will be invoked.

Note that `String` was already defined (in this case by Java itself) before we defined the protocol, and yet we were still able to successfully extend the new protocol to it. This isn't possible in many languages. For example, Java requires that you define all the method names and their groupings (known as *interfaces*) before you can define a class that implements them, a restriction that's known as the *expression problem*.

> **THE EXPRESSION PROBLEM** The expression problem refers to the desire to implement an existing set of abstract methods for an existing concrete class without having to change the code that defines either. Object-oriented languages allow you to implement an existing abstract method in a concrete class you control (interface inheritance), but if the concrete class is outside your control, the options for making it implement new or existing abstract methods tend to be sparse. Some dynamic languages such as Ruby and JavaScript provide partial solutions to this problem by allowing you to add methods to an existing concrete object, a feature sometimes known as *monkey-patching*.

A Clojure protocol can be extended to any type where it makes sense, even those that were never anticipated by the original implementor of the type or the original designer of the protocol. We'll dive deeper into Clojure's flavor of polymorphism in chapter 9, but we hope now you have a basic idea of how it works.

SUBTYPING AND INTERFACE-ORIENTED PROGRAMMING

Clojure provides a form of subtyping by allowing the creation of ad-hoc hierarchies. We'll delve into leveraging the ad-hoc hierarchy facility later, in section 9.2. Likewise, Clojure provides a capability similar to Java's interfaces via its protocol mechanism. By defining a logically grouped set of functions, you can begin to define *protocols* to which data-type abstractions must adhere. This *abstraction-oriented programming* model is key in building large-scale applications, as you'll discover in section 9.3 and beyond.

ENCAPSULATION

If Clojure isn't oriented around classes, then how does it provide encapsulation? Imagine that you need a simple function that, given a representation of a chessboard and a coordinate, returns a simple representation of the piece at the given square. To keep the implementation as simple as possible, we'll use a vector containing a set of characters corresponding to the colored chess pieces, as shown next.

Listing 1.3 A simple chessboard representation in Clojure

```
(ns joy.chess)

(defn initial-board []
  [\r \n \b \q \k \b \n \r
   \p \p \p \p \p \p \p \p          <-- Lowercase dark
   \- \- \- \- \- \- \- \-
   \- \- \- \- \- \- \- \-
   \- \- \- \- \- \- \- \-
   \- \- \- \- \- \- \- \-
   \P \P \P \P \P \P \P \P          <-- Uppercase light
   \R \N \B \Q \K \B \N \R])
```

There's no need to complicate matters with the chessboard representation; chess is hard enough. This data structure in the code corresponds directly to an actual chessboard in the starting position, as shown in figure 1.4.

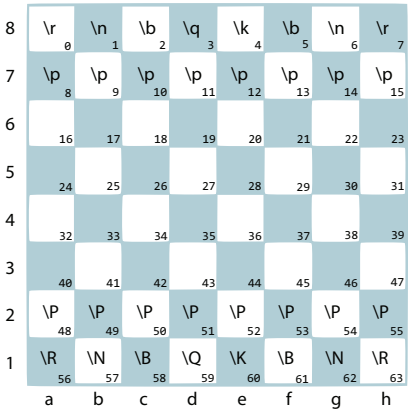

Figure 1.4
The corresponding chessboard layout

From the figure, you can gather that the black pieces are lowercase characters and white pieces are uppercase. This kind of structure is likely not optimal, but it's a good start. You can ignore the actual implementation details for now and focus on the client interface to query the board for square occupations. This is a perfect opportunity to enforce encapsulation to avoid drowning the client in board implementation details. Fortunately, programming languages with closures automatically support a form of encapsulation (Crockford 2008) to group functions with their supporting data.[8]

The functions in listing 1.4 are self-evident in their intent[9] and are encapsulated at the level of the namespace `joy.chess` through the use of the `defn-` macro that creates namespace private functions. The command for using the `lookup` function in this case would be `(joy.chess/lookup (initial-board) "a1")`.

Listing 1.4 Querying the squares of a chessboard

```
(def *file-key* \a)
(def *rank-key* \0)

(defn- file-component [file]
  (- (int file) (int *file-key*)))

(defn- rank-component [rank]
  (* 8 (- 8 (- (int rank) (int *rank-key*)))))

(defn- index [file rank]
  (+ (file-component file) (rank-component rank)))

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

Calculate file (horizontal) projection

Calculate rank (vertical) projection

Project ID layout onto logical 2D chessboard

Clojure's namespace encapsulation is the most prevalent form of encapsulation that you'll encounter when exploring idiomatic source code. But the use of lexical closures provides more options for encapsulation: block-level encapsulation, as shown in listing 1.5, and local encapsulation, both of which effectively aggregate unimportant details within a smaller scope.

---

[8] This form of encapsulation is described as the module pattern. But the module pattern as implemented with JavaScript provides some level of data hiding also, whereas in Clojure—not so much.

[9] And as a nice bonus, these functions can be generalized to project a 2D structure of any size to a 1D representation—which we leave to you as an exercise.

Listing 1.5 Using block-level encapsulation

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

It's often a good idea to aggregate relevant data, functions, and macros at their most specific scope. You'd still call `lookup` as before, but now the ancillary functions aren't readily visible to the larger enclosing scope—in this case, the namespace `joy.chess`. In the preceding code, we've taken the `file-component` and `rank-component` functions and the `*file-key*` and `*rank-key*` values out of the namespace proper and rolled them into a block-level `index` function defined with the body of the `letfn` macro. Within this body, we then define the `lookup` function, thus limiting the client exposure to the chessboard API and hiding the implementation specific functions and forms. But we can further limit the scope of the encapsulation, as shown in the next listing, by shrinking the scope even more to a truly function-local context.

Listing 1.6 Local encapsulation

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

Finally, we've now pulled *all* of the implementation-specific details into the body of the `lookup2` function itself. This localizes the scope of the `index` function and all auxiliary values to only the relevant party—`lookup2`. As a nice bonus, `lookup2` is simple and compact without sacrificing readability. But Clojure eschews the notion of data-hiding encapsulation featured prominently in most object-oriented languages.

NOT EVERYTHING IS AN OBJECT

Finally, another downside to object-oriented programming is the tight coupling between function and data. In fact, the Java programming language forces you to build programs entirely from class hierarchies, restricting all functionality to containing methods in a highly restrictive "Kingdom of Nouns" (Yegge 2006). This environment is so restrictive that programmers are often forced to turn a blind eye to awkward attachments of inappropriately grouped methods and classes. It's because of the proliferation of this stringent object-centric viewpoint that Java code tends toward being verbose and complex (Budd 1995). Clojure functions are data, yet this in no way restricts the decoupling of data and the functions that work upon them. Many of what programmers perceive to be classes are data tables that Clojure provides via

maps[10] and records. The final strike against viewing everything as an object is that mathematicians view little (if anything) as objects (Abadi 1996). Instead, mathematics is built on the relationships between one set of elements and another through the application of functions.

1.5 Summary

We've covered a lot of conceptual ground in this chapter, but it was necessary in order to define the terms used throughout the remainder of the book. Likewise, it's important to understand Clojure's underpinnings in order to frame the discussion for the rest of the book. If you've taken in the previous sections and internalized them, then congratulations: you have a solid basis for proceeding to the rest of the book. But if you're still not sure what to make of Clojure, it's okay—we understand that it may be a lot to take in all at once. Understanding will come gradually as we piece together Clojure's story. For those of you coming from a functional programming background, you'll likely have recognized much of the discussion in the previous sections, but perhaps with some surprising twists. Conversely, if your background is more rooted in object-oriented programming, then you may get the feeling that Clojure is very different than you're accustomed to. Though in many ways this is true, in the coming chapters you'll see how Clojure elegantly solves many of the problems that you deal with on a daily basis. Clojure approaches solving software problems from a different angle than classical object-oriented techniques, but it does so having been motivated by their fundamental strengths and shortcomings.

With this conceptual underpinning in place, it's time to make a quick run through Clojure's technical basics and syntax. We'll be moving fairly quickly, but no faster than necessary to get to the deeper topics in following chapters. So hang on to your REPL, here we go...

[10] See section 5.6 for more discussion on this idea.

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]