brief contents

1 ■ Introducing Clojure  1
2 ■ Clojure elements: Data structures and functions  16
3 ■ Building blocks of Clojure  55
4 ■ Multimethod polymorphism  98
5 ■ Exploring Clojure and Java interop  116
6 ■ State and the concurrent world  135
7 ■ Evolving Clojure through macros  166
8 ■ More on functional programming  186
9 ■ Protocols, records, and types  217
10 ■ Test-driven development and more  247
11 ■ More macros and DSLs  268
12 ■ Conclusion  301
Any sufficiently complicated C or Fortran program contains an ad hoc, informally specified, bug-ridden, slow implementation of half of Common Lisp.

—Philip Greenspun (http://philip.greenspun.com/research/)

1.1 Clojure: What and why?

Clojure is a simple and succinct programming language designed to leverage easily both legacy code and modern multicore processors. Its simplicity comes from a sparse and regular syntax. Its succinctness comes from dynamic typing and functions-as-values (that is, functional programming). It can easily use existing Java libraries because it’s hosted on the Java virtual machine. And, finally, it simplifies multithreaded programming by using immutable data structures and providing powerful concurrency constructs.
This book covers Clojure version 1.6. In the first few chapters you’ll learn the fundamentals of Clojure: its syntax, building blocks, data structures, Java interoperability, and concurrency features. As we progress beyond the basics, you’ll learn how Clojure can simplify larger programs using macros, protocols and records, and higher-order functions. By the end of this book you’ll understand why Clojure is traveling a rare path to popularity and how it can transform your approach to developing software.

Clojure’s strengths don’t lie on a single axis. On the one hand, it’s designed as a hosted language, taking advantage of the technical strengths of platforms like the JVM, Microsoft’s Common Language Runtime (CLR), and JavaScript engines on which it runs, while adding the “succinctness, flexibility, and productivity” (http://clojure.org/rationale) of a dynamically typed language. Clojure’s functional programming features, including high-performance immutable data structures and a rich set of APIs for working with them, result in simpler programs that are easier to test and reason about. Pervasive immutability also plays a central role in Clojure’s safe, well-defined concurrency and parallelism constructs. Finally, Clojure’s syntax derives from the Lisp tradition, which brings with it an elegant simplicity and powerful metaprogramming tools (http://clojure.org/rationale).

Some of these points may elicit an immediate positive or negative reaction, like whether you have a preference for statically or dynamically typed languages. Other language design decisions may not be entirely clear. What is a functional programming language and is Clojure like other ones you may have seen? Does Clojure also have an object system or provide design abstractions similar to mainstream object-oriented (OO) languages? What are the advantages and disadvantages of hosting the language on an existing VM?

The promise of Clojure’s synthesis of features is a language that’s composed of simple, comprehensible parts that not only provide power and flexibility to writing programs but also liberate your understanding of how the parts of a language can fit together. Let no one deceive you: there are many things to learn. Developing in Clojure requires learning how to read and write Lisp, a willingness to embrace a functional style of programming, and a basic understanding of the JVM and its runtime libraries. We’ll introduce all three of these Clojure pillars in this chapter to arm you for what lies ahead in the rest of the book: a deep dive into an incredible language that’s both new and old.

1.1 Clojure: A modern Lisp

Clojure is a fresh take on Lisp, one of the oldest programming language families still in active use (second only to Fortran). Lisp isn’t a single, specific language but rather a style of programming language that was designed in 1958 by Turing award winner John McCarthy. Today the Lisp family consists primarily of Common Lisp, Scheme, and Emacs Lisp, with Clojure as one of the newest additions. Despite its fragmented history, Lisp implementations, including Clojure, are used for cutting-edge software systems in various domains: NASA’s Pathfinder mission-planning software, algorithmic
trading at hedge funds, flight-delay prediction, data mining, natural language processing, expert systems, bio-informatics, robotics, electronic design automation, web development, next-generation databases (http://www.datomic.com), and many others.

Clojure belongs to the Lisp family of languages, but it doesn’t adhere to any existing implementation exclusively, preferring instead to combine the strengths of several Lisps as well as features from languages like ML and Haskell. Lisp has the reputation of being a dark art, a secret weapon of success, and has been the birthplace of language features like conditionals, automatic garbage collection, macros, and functions as language values (not just procedures or subroutines; http://paulgraham.com/lisp.html). Clojure builds on this Lisp tradition with a pragmatic approach to functional programming, a symbiotic relationship with existing runtimes like the JVM, and advanced features like built-in concurrency and parallelism support.

You’ll get a practical sense of what it means for Clojure to be a Lisp when we explore its syntax later in this chapter, but before we get bogged down in the details, let’s consider the other two pillars of Clojure’s design: Clojure as a functional programming language hosted on the JVM.

### 1.1.2 Clojure: Pragmatic functional programming

Functional programming (FP) languages have seen an explosion in popularity in the last few years. Languages like Haskell, OCaml, Scala, and F# have risen from obscurity, and existing languages like C/C++, Java, C#, Python, and Ruby have borrowed features popularized by these languages. With all of this activity in the community, it can be difficult to determine what defines a functional programming language.

The minimum requirement to be a functional language is to treat functions as something more than named subroutines for executing blocks of code. Functions in an FP language are *values*, just like the string "hello" and the number 42 are values. You can pass functions as arguments to other functions, and functions can return functions as output values. If a programming language can treat a function as a value, it’s often said to have “first-class” functions. All of this may sound either impossible or too abstract at this point, so just keep in mind that you’re going to see functions used in new, interesting ways in the code examples later in this chapter.

In addition to functions as first-class values, most FP languages also include the following unique features:

- Pure functions with referential transparency
- Immutable data structures as the default
- Controlled, explicit changes to state

These three features are interrelated. Most functions in an FP design are *pure*, which means they don’t have any side-effects on the world around them such as changing global state or doing I/O operations. Functions should also be *referentially transparent*, meaning that if you give the same function the same inputs, it will always return the same output. At the most elementary level, functions that behave this way are
simple, and it’s simpler and easier\textsuperscript{1} to reason about code that behaves consistently, without respect to the implicit environment in which it runs. Making immutable data structures the language default guarantees that functions can’t alter the arguments passed to them and thus makes it much easier to write pure, referentially transparent functions. In a simplistic sense, it’s as if arguments are always passed by value and not by reference.

“Hold on,” you might say, “passing arguments by value and copying data structures everywhere is expensive, and I need to \textit{change} the values of my variables!” Clojure’s immutable data structures are based on research into the implementation of performant, purely functional data structures designed to avoid expensive copying.\textsuperscript{2} In theory, if you make a change to an immutable data structure, that change results in a brand-new data structure, because you can’t change what’s immutable. In reality, Clojure employs \textit{structural sharing} and other techniques under the hood to ensure that only the minimum amount of copying is performed and that operations on immutable data structures are fast and conserve memory. In effect, you get the safety of passing by value with the speed of passing by reference.

Persistent data structures can’t be changed, but the diagrams in figures 1.1 and 1.2 demonstrate how one might “edit” a persistent tree. The tree \textit{xs} shown in figure 1.1 consists of immutable nodes (circled letters) and references (arrows), so it’s impossible to add or remove a value from tree \textit{xs}. But you could create a new tree that shares as much of the original tree \textit{xs} as possible. Figure 1.2 demonstrates how you can add a new value \textit{e} by creating a new set of nodes and references in the path to the root of the tree (\textit{d’}, \textit{g’}, \textit{f’}) that reuse old nodes (\textit{b}, \textit{a}, \textit{c}, and \textit{h}), resulting in the new persistent tree \textit{ys}. This is one of the basic principles underlying Clojure’s persistent data structures.

\textsuperscript{1} See the talk “Simplicity Ain’t Easy” to understand the unique role \textit{simplicity} has in Clojure’s design considerations: http://youtu.be/cidchWg74Y4. For a deeper but more abstract and less Clojure-centric presentation of the easy-versus-simple distinction, watch “Simple Made Easy” by Clojure’s creator Rich Hickey: http://www.infoq.com/presentations/Simple-Made-Easy.

Things in your programs change, though. Most programming languages have variables that serve as named pieces of state that you can change at any time. In Clojure, the story is more controlled and better defined. As a fact, values like the number 42 can’t change; 42 is 42, and subtracting 2 from 42 doesn’t change the number 42 but rather gives a new value of 40. This truth extends to all values, not just numbers. On the other hand, if you have a variable acting as the identity for something in your program that has the value 42 initially assigned to it, you might want to assign a new value to that variable at some later point in your program. In this case a variable is like a container into which you may put different values at different times. In a multi-threaded, concurrent world, your programming language should provide you assurances about how those changes take place, and Clojure does just that.

Clojure lets you change the values variables hold but with well-defined semantics regarding how and when the changes take place. If you have one variable and you want to change its value, Clojure lets you do that atomically, so you’re certain that if multiple threads of execution are looking at a variable’s value, they always get a consistent picture, and that when it changes it does so in a single, atomic operation. If you need to change multiple variables together as a unit, Clojure has a separate facility using its software transactional memory (STM) system to change multiple variables as part of a transaction and rollback changes if they don’t all complete as expected. If you need to change a variable but want that change to happen on a separate thread of execution so it doesn’t block the main thread of your program, Clojure provides facilities for that as well. All of these are built into the core of the

---

3 In this case, “atomic” is a synonym for “indivisible.” If an operation is atomic, then no other operations can interfere with the underlying state while it’s being changed. If any other processes attempt to get the state of a variable during an atomic operation, they simply get the last value of the variable before the atomic operation began. In the case of other processes attempting to change the underlying state during an atomic operation, they’re held off until the atomic operation is complete.
language, making concurrency so easy you have to work to make your programs not support it.\(^4\)

Functional languages are often judged by their functional “purity,” or strict adherence to the theoretical underpinnings of functional programming language design. On the one hand, Clojure’s default use patterns encourage pure functional programming: immutable data structures, higher-order functions and recursion that take the place of imperative loops, and even a choice between lazy or eager evaluation of collections. On the other hand, Clojure is pragmatic. Even though most problems can be solved using immutable data structures and functional programming patterns, certain tasks are more clearly modeled with mutable state and a more imperative approach. Clojure provides constructs with well-defined semantics for sharing state and changing it over time as we’ve just described. In addition, Clojure also doesn’t require the developer to annotate code that causes side-effects, whether they be changes to state, printing to the screen, or network I/O, as some “purer” functional programming languages require.

Another part of Clojure’s pragmatism stems from its hosted design. When necessary, you can always drop down to the host platform and use Java APIs directly from Clojure, with all of the performance (and pitfalls) that come from coding directly in Java.

### 1.1.3 Clojure on the JVM

Clojure was designed as a *hosted* language. Whereas most programming language projects combine a language design with an accompanying runtime platform for that language, Rich Hickey, Clojure’s creator, decided to focus on Clojure-the-language and rely on existing VMs for the runtime platform. He began his work on the JVM, but Clojure has since spread to the CLR with interoperability with the .NET ecosystem (Clojure-CLR), as well as to browser and server-side JavaScript engines (ClojureScript).

Rich made this decision with the best kind of engineering laziness in mind ([http://blog.codinghorror.com/how-to-be-lazy-dumb-and-successful/](http://blog.codinghorror.com/how-to-be-lazy-dumb-and-successful/)). The JVM is a mature, ubiquitous platform with a myriad of third-party libraries. The canonical HotSpot JVM implementation is open source and sports an advanced just-in-time (JIT) compiler with choice of garbage collectors, maintaining competitive performance with “native” runtimes for a variety of use cases.\(^5\) By taking these features for granted as part of the underlying runtime host, the Clojure community is free to focus its time on a solid language design and higher-level abstractions instead of reinventing the VM wheel (and the bugs that come with it).

From a business perspective, relying on existing VMs lowers the risk of introducing Clojure. Many organizations have existing architectures and personnel expertise tied

---

\(^4\) For those already familiar with Clojure, note that we use the term *variable* loosely at this point to introduce Clojure’s unique handling of values, identities, and underlying state and how those all change over time. We’ll cover the specifics of Clojure’s concurrency constructs in a later section using Clojure’s precise terminology.

to the JVM or the CLR and the ability to introduce Clojure as part of a larger Java or C# application is a powerful selling point. Clojure compiles down to bytecode on the JVM and Common Intermediate Language (CIL) on the CLR, meaning that it participates as a first-class citizen of the VMs it runs on.

On the other hand, Clojure intentionally doesn’t shield you from the host platform on which it runs. To be effective in Clojure on the JVM, you’ll have to learn about its runtime environment, including the following at a minimum:

- Java’s core `java.lang.*` classes and their methods
- The JVM’s threading/process model
- How the JVM finds code to compile on its `classpath`

We’ll introduce these minimum Java and JVM concepts in this chapter and more advanced topics as we encounter them, so you don’t need to put this book down and study Java first. If you’re interested in working with Clojure on the CLR or a JavaScript engine, you’ll need to have an equivalent understanding of those platforms to use Clojure on them effectively.

Now that you have a high-level understanding of Clojure as a functional Lisp on the JVM, let’s get started writing some Clojure code to bring these concepts to life.

### 1.2 Language basics

It’s impossible to separate the Lisp, functional programming, and JVM features of Clojure. At every step they play on each other and tell a compelling software development story, but because we have to start somewhere, tackling the syntax on the page is a good place to start.

#### 1.2.1 Lisp syntax

Clojure’s syntax is derived from its Lisp roots: lots of parentheses. It’s alien to most developers with experience in languages with Algol-inspired syntax like C, C++, Java, Python, Ruby, Perl, and so on. Because it’s so foreign, there are some tricks we’ll employ for getting over the parenthesis hump:

- Initially ignore the parentheses.
- Consider how other languages use parentheses.
- See parentheses as “units of value” or *expressions*.
- Embrace the parentheses.

To convince you that initially ignoring the parentheses is okay, let’s take our first look at some example code:

```clojure
(get-url "http://example.com")
```

If you guessed that this makes an HTTP request for the URL http://example.com, you’re correct. The `get-url` function isn’t defined in Clojure by default, but it makes for a nice self-describing function name, and we’ll use this as one of our main examples.
once we get past the basics. Let's look at some code examples that use built-in Clojure functions and see their output:

```clojure
(str "Hello, " "World!"
;; Result: "Hello, World!"
;; (A Semi-colon starts a code comment which continues
;; to the end of the line.)
```

The `str` function stands for “string” and concatenates its arguments into a single output string. Other languages generally use an operator like `+`. What would it look like to concatenate several strings with operators?

"Hello from " + "a language " + "with operators";
# Result: "Hello from a language with operators"

This is called *infix notation* because you put the operator *in between* each string that you’re concatenating. As a Lisp, Clojure uses *prefix notation* for all of its functions and even things that look like operators, so if you need to concatenate more than two strings, you just keep passing arguments to the `str` function:

```clojure
(str "Hello from " "Clojure with " "lots of " "arguments")
;; Result: "Hello from Clojure with lots of arguments"
```

And if you need to do arithmetic, the same principle holds:

```clojure
(+ 1 2)
;; Result: 3
(+ 1 2 3)
;; Result: 6
```

These examples demonstrate two advantages of Clojure’s approach. First, there’s no difference between functions and operators because Clojure *doesn’t have operators*. There’s no system of operator precedence to memorize. The forms `str` and `+` are both regular Clojure functions; one just happens to have a nonalphabetic character as its name. Second, because you don’t need to interleave operators between arguments, it’s natural for these kinds of functions to take an arbitrary number of arguments (called *variable arity*), allowing you to add more arguments without fear of forgetting to put an operator in between each one.

In the preceding examples, you can safely ignore the parentheses, but let’s step up the difficulty. If you needed to do more than one operation, you might write the following in a language that uses operators for arithmetic:

```
3 + 4 * 2
```

In a language with operators you’d need to remember the precedence of the `+` and `*` operators, but you can make this unambiguous regardless of language by surrounding expressions with parentheses:

```
3 + (4 * 2)
# Result: 11
```
By not having operators, Clojure makes that level of explicitness a requirement:

```
(+ 3 (* 4 2))
;; Result: 11
```

Let’s break that down expression by expression.

The outermost function is `+`, which has two arguments: 3 and the form `(* 4 2)`. You
know what 3 is all about, so let’s solve the `(* 4 2)`. If you call the multiplication function `*` with arguments of 4 and 2, you get 8. Let’s write the expression again, solving
the `(* 4 2)` step first, bolding the important parts to call your attention to them:

```
(+ 3 (* 4 2))
(+ 3 8)
;; Result: 11
```

Now you have the `+` function with two simple arguments and the sum is obviously 11.
Although leveraging operators and their precedence rules makes writing such mathe-
matical expressions more concise in other languages, Clojure makes calling functions
completely consistent across the language.

Now that you’ve seen some parentheses in action, let’s stop ignoring them for a
moment and understand their primary purpose.

1.2.2 Parentheses

Lisp’s use of parentheses is its secret syntactic weapon, but we’re not going to delve
into their deeper purpose right now. For the sake of reading and writing your first
Clojure programs, we’re going to say that parentheses serve two purposes:

- Calling functions
- Constructing lists

All of the code so far has shown examples of the first pur-
pose—to call functions. Inside a set of parentheses, the first
language form is always a function, macro, or special form,
and all subsequent forms are its arguments. Figure 1.3 is a
simple example of this use of parentheses. We’ll cover
what macros and special forms are as we encounter them,
but for now you can think of them as functions that get
special treatment.

Start training your brain to associate left parenthesis
with function invocation. That left parenthesis is like a
phone being held up to the function’s ear, getting ready to call it with the rest of the
items up to the matching right parenthesis. It will become increasingly important to
have this association firmly planted in your mind once we start looking at higher-order
functional programming patterns. Also remember that arguments to functions won’t
always be simple values but, as in the earlier examples, will be nested expressions—see
figure 1.4 for an example.
The second use of parentheses is at once the most common and the least noticeable—to construct lists. On the one hand, Clojure has literal syntax for collections other than lists, and idiomatic Clojure programs use all of the collection types based on their different performance strengths. Clojure isn’t as list-centric as other Lisps, in part because it provides literal syntax for these other types of collections. On the other hand, at the meta level, your entire Clojure program is a series of lists: the very source code of your program is interpreted by the Clojure compiler as lists that contain function names and arguments that need to be parsed, evaluated, and compiled. Because the same language features are available at both the lower compiler levels and in your normal program code, Lisp enables uniquely powerful meta-programming capabilities. We’ll delve into the significance of this fact when we discuss Clojure macros in later chapters, but for now let’s take a tour of Clojure’s essential data structures and collection types so we can read realistic code examples.

You’ve now seen Clojure’s essential syntax: parentheses that contain functions (or special things that act like functions) and their arguments. Because parentheses are the containers for all expressions in the language, you edit your Clojure code by arranging these expressions like building blocks, each one a little self-contained world of functionality that results in a consistent value and can be placed anywhere in your program where that value is required. Moreover, this consistency in parenthetical syntax means IDEs and text editors can provide structural editing for moving expressions around easily, meaning you never have to make sure your open and close parentheses are matched. As you spend more time writing Clojure, we highly recommend learning some of these tools in your development environment of choice, so that Clojure’s parentheses become an advantage and not a hindrance.

1.3 Host interoperation: A JVM crash course

Clojure doesn’t hide the host platform on which it’s implemented from the programmer. In this book we’ll focus on the canonical JVM implementation of Clojure, but the principles of interoperation with the host, usually just called interop, are common to all of the platforms that Clojure targets. Because Clojure embraces its host platform instead of trying to hide it, you must learn the basics of Java and the JVM to be able to code in Clojure.

Java is three distinct pieces that were designed and shipped together: a language, a virtual machine, and a standard library. Parts of Clojure are written in Java the language, but Clojure itself doesn’t use it. Instead, Clojure code is compiled directly to
bytecode for the JVM to run. Clojure also requires you to use the standard library for
many basic functions. Because the standard library was written in and for Java the lan-
guage, some basic knowledge of Java the language will help you make better use of
Java the library.

In many cases Clojure uses Java types and the standard library directly. For example,
strings in Clojure are Java String objects and literal numerals are Java Long objects, and
Clojure’s collections implement the same collection interfaces implemented by Java
collections. Reusing Java types and interfaces has the added benefit that Java code can
use Clojure types (such as its immutable data structures) seamlessly.

Sometimes Clojure wraps Java library features with functions of its own, like many
of the functions in Clojure’s clojure.string namespace that delegate to methods in
Java’s String class. But often there’s no Clojure wrapper and you’ll need to call Java
methods directly. For example, Clojure doesn’t implement regular functions for
mathematical methods like abs[olute], exp[onent], log, sin, cos, and tan found in the
java.lang.Math class, which therefore need to be invoked via the Java interop syntax
we introduce later in this section.

Let’s briefly review Java’s types, classes, and object system, so that we can make
sense of what it means for Clojure code to interoperate with Java.

1.3.1 Java types, classes, and objects

Java is an object-oriented language based on a class hierarchy with single inheritance.
In addition to classes, common behaviors can be grouped into interfaces, which act as
simple outlines of method signatures that classes that implement the interface must
support. Only one public class or interface can be defined in a single file, and those
files must be placed in directories that are on Java’s classpath. The Java classpath is
akin to C’s search path or Ruby’s $LOAD_PATH in that it’s a collection of directories that
the Java compiler will search when looking for files to compile as part of your pro-
gram. The fully qualified name of a Java class or interface consists of its package name
followed by the name of the class or interface being defined; for example, Java’s Math
class is situated in the java.lang package. This allows individual classes to share the
same name (for example, Math) as long as they aren’t in the same package and thus
have a unique full name when loaded into the JVM (for example, java.lang.Math vs.

What does all of this have to do with Clojure? All of the classes located in Java’s
java.lang package are imported by default in all Clojure programs, so that you can refer
to things like String and Integer without having to type out java.lang.String and
java.lang.Integer. Many Clojure data structures (especially collections) implement

---

6 API documentation for the Math class can be found at http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html.
7 Java 8 introduced default methods for interfaces. Because, as of this writing, Clojure currently targets Java 6
as its minimum target version, we’ll continue to treat interfaces as simple method contracts without default
implementations.
Java interfaces, so Java libraries that expect objects that implement those interfaces will accept Clojure data structures as arguments. All Clojure collections, for example, implement java.lang.Iterable or java.util.Collection, whereas only some implement java.util.List or java.util.Map, depending on their purpose.

Like the Java compiler, the Clojure compiler expects to find your Clojure source code on the Java classpath and also expects that the full name of a namespace be unique. Appendix A covers the particulars of how projects are organized on the file system, how the classpath is set up, and how to invoke the Clojure compiler.

The flip side to having to learn some Java basics is that you get access to a plethora of mature, battle-tested Java libraries that you can consume seamlessly from your Clojure programs: Joda Time provides correct date and time manipulation; JDBC drivers expose a common API for communicating with different databases; Jetty is an advanced embeddable web server; Bouncy Castle has a convenient API for working with Java’s cryptographic features; Selenium WebDriver lets you test web applications by controlling real web browsers programmatically; and the various Apache Commons libraries provide miscellaneous utilities that act as an extended Java standard library. In addition to application libraries, you can use all of the built-in tools for monitoring the performance of the JVM, as well as external profilers like VisualVM, YourKit, and profilers-as-a-service like New Relic to gain a deeper understanding of how your Clojure applications run.

Having described all the wonderful features you have access to via Java interop, we still haven’t discussed how to access them from Clojure. How does Clojure differentiate between regular Clojure code and code that does Java interop? The first part of this answer is the dot operator.

### 1.3.2 The dot and new operators

The dot operator—written as a literal .—forms the basis for Java interop. When seen by itself after an opening parenthesis, it should be read as “in the scope of A do B with arguments....” For example:

```
(. Math PI) ;; Result: 3.141592653589793
(. Math abs -3) ;; Result: 3
(. "foo" toUpperCase) ;; Result: "FOO"
```

To accommodate the fact that, outside of Java interop, the first form in a Clojure expression is a function, macro, or special form, Clojure provides some syntactic sugar to make this code more idiomatic.

The first two examples deal with static members of the Math class and can be rewritten like this:

```
Math/PI;; Result: 3.141592653589793
(Math/abs -3)
;; Result: 3
```
Fields and methods that are static (defined on the class and not on instances of the class) are accessed with a forward slash. In the Java Math class, PI is a static field and not a method, so it doesn’t need to be invoked (using parentheses) to return a value. But abs is a method, so it must still be invoked with parentheses.

The third example is an instance method invocation: it calls the toUpperCase method of the string instance "foo". This example can be rewritten as follows to make it look more like a function call:

```
(.toUpperCase "foo")
;; Result: "FOO"
```

To create instances of classes, you can use the new operator or a trailing dot to indicate that the class’s constructor should be called:

```
(new Integer "42")
;; Result: 42
(Integer. "42")
;; Result: 42
```

The trailing dot here, the leading dot for instance fields and methods, and the forward slashes for static fields and methods are all syntactic conveniences. During Clojure’s macro expansion phase of compiling code, the trailing dot expands to use the new special form, and the others expand to the standalone dot form demonstrated at the beginning of this section, so they’re all literally equivalent by the time your Clojure code is evaluated.

The dot operator only provides a doorway for consuming Java APIs. We’ll cover other advanced topics of Java interop that involve extending Java’s class system in later chapters. More importantly, we’ll explore the powerful design abstractions that Clojure provides in spite of the underlying object-oriented nature of its host platform as we work through programs in later chapters. Before closing out this chapter, we should touch on one more aspect of the JVM that’s central to Clojure’s mission: the JVM model for threads and concurrency.

### 1.3.3 Threads and concurrency

A thread represents program execution. Every program, regardless of programming language, has at least one main thread or process in which the application code is being evaluated. In addition to this main application thread, language runtimes generally provide a way to start new, separate threads of execution. The default runtimes for Ruby and Python, for example, provide lightweight or "green" threads that are managed completely by the runtime itself. JVM threads map directly to native system threads, which means they can take advantage of multiple CPU cores “for free” by letting the operating system manage scheduling threads and delegating to CPUs. By engaging all available cores on a machine using native threads, the JVM provides genuine and performant parallelism.
In an application with a single thread of execution, the program is evaluated serially, and it’s relatively simple to understand when objects are created, changed, and destroyed based on the flow of the program. But when introducing additional threads of execution that are running at the same time as the main thread, issues of concurrency have to be dealt with. If you have state (a variable) that can be accessed from multiple threads simultaneously, how can you be sure that two threads aren’t attempting to make changes to that state at the same time? Are you certain that changes to the state can be performed atomically, such that no other threads see a corrupt “in progress” state for a variable that’s being changed during a program’s execution?

Although Java has all the tools necessary to write safe concurrent programs with shared mutable state, in practice it’s extremely difficult to write such programs correctly. Having spent years writing such programs himself in Java and other languages, Rich Hickey implemented a set of concurrency constructs in Clojure that not only allow for correctness but also enforce it at the language level.

By virtue of the fact that Clojure’s core data structures are all immutable, the issue of shared mutable state becomes largely moot. When some mutable state is required, Clojure provides concurrency data structures called vars, atoms, refs, and agents that have clearly defined semantics for how to change the underlying state they reference. Furthermore, Clojure always allows fast access to the value of these data structures—even if they’re in the middle of being changed by another thread—by maintaining a snapshot of the old values during changes. For use cases that need only parallel execution but not shared state, Clojure provides futures and promises similar to other languages but implemented using JVM threads and not bound to any particular callback as is common in languages like JavaScript.

Instead of comparing all of the specific features of each concurrency construct in the abstract or listing all the functions that deal with parallel execution here, we’ll continue exploring these topics in more depth with code examples in later chapters.

1.4 Summary

We’ve completed our transit around the three basic pillars of Clojure: functional programming with immutable data structures, Lisp syntax, and host interop. You now know the absolute basics of reading Lisp and Java interop code, and we can continue to explore Clojure’s functions and data structures and worry about the underlying platform only when the need arises.

The most difficult aspect of learning Clojure for most developers isn’t the syntax of Lisp or the idiosyncrasies of the JVM platform. The truly mind-bending part of coding in Clojure comes from the shift from an imperative mindset (prevalent in most mainstream languages) to a functional programming approach to program design. Much of your time in the beginning will be spent wondering how to accomplish things in Clojure that you can already do easily in your imperative language of choice. In our experience, people often spend more time unlearning complex idioms from other
languages than they do learning the simple, flexible, and composable parts that make up the Clojure language.

The chapters that follow will cover Clojure’s core data structures and APIs in detail using code examples that demonstrate the strengths of Clojure’s language design. By the end of the next chapter, you’ll have the skills to set up your own Clojure projects and write small- to medium-size programs.
Clojure is a modern Lisp for the JVM. It has the strengths you expect: first-class functions, macros, and Lisp’s clean programming style. It supports functional programming, making it ideal for concurrent programming and for creating domain-specific languages. Clojure lets you solve harder problems, make faster changes, and end up with a smaller code base. It’s no wonder that there are so many Clojure success stories.

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

What’s Inside

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

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

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

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

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

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

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

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

$49.99 / Can $57.99 [INCLUDING eBook]