Clojure Distilled

The difficulty in learning Clojure does not stem from its syntax, which happens to be extremely simple, but from having to learn new methods for solving problems. As such, we'll focus on understanding the core concepts and how they can be combined to solve problems the functional way.

All the mainstream languages belong to the same family. Once you learn one of these languages there is very little effort involved in learning another. Generally, all you have to do is learn some syntax sugar and the useful functions in the standard library to become productive. There might be a new concept here and there, but most of your existing skills are easily transferrable.

This is not the case with Clojure. Being a Lisp dialect, it comes from a different family of languages and requires learning new concepts in order to use effectively. There is no reason to be discouraged if the code appears hard to read at first. I assure you that the syntax is not inherently difficult to understand, and that with a bit of practice you might find it to be quite the opposite.

This goal of this guide is to provide an overview of the core concepts necessary to become productive with Clojure. Let's start by examining some of the key advantages of the functional style and why you would want to learn a functional language in the first place.

Immutable

Clojure is a functional language. This makes it extremely well positioned for writing large applications. As the application grows it's imperative to be able to reason about its constituent parts in isolation. Conversely, there is a lot of value in building code out of components that are testable and reusable by nature. Let's take a look at why the functional approach is such a good fit here.

Functional languages are ideal for writing large applications because they eschew global state and favor immutability as the default. Having immutable shared state allows us to safely reason about parts of the application in isolation. At first glance, the idea of using immutable data structures sounds unnecessarily restrictive. However, as we'll soon see, many of the benefits associated with the functional style stem directly from it.

Mutable data can either be passed around by value or by reference. The safe approach would be to pass the data by value as it guarantees that any changes to the data will remain in the local scope. Unfortunately, this approach is extremely inefficient, so most languages pass data by reference instead.

Passing data by reference is fast, but it makes the code difficult to reason about. In order to safely work with the data you have to know all the places where it might be referenced. The complexity grows with the size of the application. The more code has access to a piece of data the more proverbial balls you end up having to juggle in your head.

Immutable data structures provide us with an ingenious alternative to the above dilemma. Every time a change is made to the data structure a revision is created. We now have the same guarantees offered by naive copying of the data, but we only pay the price proportional to the size of the change.

Just like garbage collection frees us from having to manually track data allocation and deallocation, immutable data structures free us from having to manage data references by hand. From the user perspective we simply "copy" the data any time we make a change. The language will take care of figuring out what parts of it can be cleaned up when they're no longer used.

Having such data structures facilitates writing pure functions. A pure function is simply a function without side effects. These functions do not rely on any state outside their inputs and they do not modify any external state when they run. Given the same parameters, the function will always produce the same result, regardless of the global state of the application.

Such functions can be safely reasoned about in isolation because we can guarantee that they're only able to modify their local scope. They provide us with self-contained components that can be composed to create complex behaviors. This type of code is referred to as being referentially transparent.

Reusable

Object-oriented languages use classes for composition. The data in each class is tightly coupled to the logic associated with it. Each class represents a specific domain and the methods written in it are not easily reusable outside that domain. When we wish to reuse the existing code we often have to resort to patterns such as adapters and wrappers.

The focus, in such a language, is primarily on modeling the state using classes. The data is seen as being incidental to the whole process. Functional programming brings data to the forefront and it encourages us to think about our problems in terms of data transformations.

In a functional language, the logic and the data are kept separate. Clojure provides a small set of common data structures such as lists, vectors, maps, and sets. All the functions in the language operate on the same data structures allowing us to combine them without any additional ceremony. With this approach the function becomes the core reusable component.

Each function represents a certain transformation that we wish to apply to our data. When we need to solve a problem we simply have to break it up into a sequence of transformations and map those to the appropriate functions. The functions capture how the tasks are accomplished, while their composition states what is being accomplished. Code that separates what is being done from how it is done is referred to as being declarative.

Let's take iteration as an example. With the imperative style we would write a loop and put the logic that's invoked during each step inside it. By contrast, the functional approach is to use an iterator function and pass the logic that we want to execute during the iteration as a parameter.

An iterator function can be written once and it encapsulates all the logic required for iteration, edge cases, and boundary checks. We can now reuse this function without having to worry about remembering to do these checks each time we need to iterate.

Scalable

The focus on immutability makes it much easier to tackle the difficult problems of parallelism and concurrency. While there is no silver bullet for addressing either problem, the language can go a long way in helping us reason about them.

As you'll recall, pure functions rely solely on their arguments and do not modify any state outside their scope. These properties make it possible to safely run them in parallel allowing us to easily take advantage of the extra cores.

An example of this is mapping a function over the items in a collection. We can start by using the `map` function. This function will iterate over a collection and apply a transformer function to each element inside it. Should we discover that each operation takes a significant amount of time, we can then simply switch to using the `pmap` function to run these operations in parallel.

Finally, it turns out that the immutable data structures are also an excellent tool for managing shared mutable state. Clojure provides a Software Transactional Memory (STM) library based on these data structures. With transactional memory we no longer have to worry about manual locking when dealing with threads. Better still, shared state backed by immutable data only needs to be locked for writing since the current revision can be safely read while the new revision is being generated.

Clojure In Action

Now that we've discussed some of the reasons to start using the functional style, let's see how to apply these ideas in practice with Clojure.

The Core

Data Types

Clojure provides a number of standard data types, most of which should look familiar:

- Vars provide mutable storage locations. These can be bound and rebound on a per-thread basis.
- Booleans can have a value of true or false; nil values are also treated as false.
- Numbers can be integers, doubles, floats, and fractions.
- Symbols are used as identifiers for variables.
- Keywords are symbols that reference themselves and are denoted by a colon; these are often used as keys in maps.
- Strings are denoted by double quotes and can span multiple lines.
- Characters are denoted by a forward slash.
- Regular expressions are strings prefixed with a hash symbol.

In addition to the data types, Clojure provides us with a literal notation for common collection types such as lists, vectors, maps, and sets:

- List: `'(1 2 3)`
- Vector: `[1 2 3]`
- Map: `{:foo "a" :bar "b"}`
- Set: `#{"a" "b" "c"}`

Interestingly, Clojure logic is written using its data structures. Using the same syntax for both data and logic allows for powerful metaprogramming features. We can manipulate any piece of Clojure code just like we would any other data structure. This feature makes it trivial to template the code for recurring patterns in your problem domain. In Clojure, code is data and data is code.

Special Forms

Special forms provide a small set of primitives, such as the `if` conditional, that define the core syntax. Majority of the language is implemented using functions and macros in the standard library.

Functions

Function calls in Clojure work the same as any mainstream languages. The main difference being that the function name comes after the paren in the Clojure version.

```
functionName(param1, param2)
```

```
(function-name param1 param2)
```

There is a very simple reason for this difference. The function call is simply a list containing the function name and its parameters. In Clojure, a list is a special type of data structure reserved for creating callable expressions. To create a list data structure we'd have to call the list function:

```
(list 1 2 3)
```

Anonymous Functions

As the name implies, anonymous functions are simply functions that aren't bound to a name. Let's take a look at the following function that accepts a single argument and prints it:

```
(fn [arg] (println arg))
```

The function is defined by using the `fn` form followed by the vector containing its argument and the body. We could call the above function by setting it as a first item in a list and its argument as the second:

```
((fn [arg] (println arg)) "hello")
=>"hello"
```

Clojure provides syntactic sugar for defining anonymous functions using the # notation. With it we can rewrite our function more concisely as follows:

```
#(println %)
```

Here, the `%` symbol indicates an unnamed argument. Multiple arguments would each be followed by a number indicating its position as seen below:

```
#(println %1 %2 %3)
```

This type of function is useful when you need to perform a one-off computations that don't warrant defining a named function. They are commonly used in conjunction with the higher-order functions that we'll see in a moment.

Named Functions

Named functions are simply anonymous functions bound to a symbol used as an identifier. Clojure provides a special form called `def` that's used for creating global variables. It accepts a name and the body to be assigned to it. We can create a named function using the `def` form as follows:

```
(def double
  (fn ([x] (* 2 x))))
```

Since this is such a common operation, Clojure provides a special form called `defn` that does it for us:

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

The `defn` from behaves the same as the `fn` form we saw above, except that its first argument is the name of the function. The body of the function can consist of multiple expressions:

```
(defn bmi [height weight]
  (println "height: height)
  (println "weight:" weight)
  (/ weight (* height height)))
```

Here we define a function to calculate the BMI using the height and weight parameters. The body consists of two print statements and a call to divide the weight by the square of the height. All the expressions are evaluated from the inside out. In the last statement, `(* height height)` is evaluated, then the weight is divided by the result and returned. In Clojure, mathematical operators, such as `/` and `*`, are regular functions and so we call them using the prefix notation as we would with any other function.

Note that only the result from the last expression is returned from the function, the results of all the other expressions are discarded. Therefore, any intermediate expressions should strictly be used for side effects as is the case with the `println` calls above.

One thing to note is that Clojure uses a single pass compiler. For this reason, the functions must be declared before they are used. In a case when we need to refer to a function before it's been declared, we must use the `declare` macro in order to provide a forward declaration.

```
(declare down)

(defn up [n]
  (if (< n 10)
    (down (+ 2 n)) n))

(defn down [n]
  (up (dec n)))
```

A keen reader will have noticed that the code is structured as a tree. This tree is called the abstract syntax tree, or AST for short. By being able to see the AST directly, we can examine the relationships between pieces of logic visually.

Since we write our code in terms of data, there are fewer syntactic hints than in most languages. For example, there is no explicit return statement and the last expression of the function body is returned implicitly.

This might take a little getting used to if you're accustomed to seeing a lot of annotations in your code. To aid readability, functions are often kept short while indentation and spacing are used for grouping code visually.

In Clojure, there is no distinction between functions and variables. You can assign a function to a label, pass it as a parameter, or return a function from another function. Functions that can be treated as data are referred to as being first-class because they don't have any additional restrictions attached to them.

Higher-Order Functions

Functions that take other functions as parameters are called higher-order functions. One example of such a function is `map`:

```
(map #(* % %) [1 2 3 4 5]) => (1 4 9 16 25)
```

This function accepts two parameters where the first is an anonymous function that squares its argument and the second is a collection of numbers. The map function will visit each item in the collection and square it.

One major advantage of using a higher order function is that we can infer the intent of the code from the function being used. Let's contrast the above example to an imperative style loop:

```
(loop [[n & numbers] [1 2 3 4 5]
       result []]
  (let [result (conj result (* n n))]
    (if numbers
      (recur numbers result)
      result)))
```

The looping approach ends up having a lot more noise and thus we have to read through the code more carefully to tell what it's doing. The other problem is that the code becomes monolithic and no part of it can be used individually.

Another example of a higher-order function is `filter`. This function goes through a collection and keeps only the items matching the specified predicate.

```
(filter even? [1 2 3 4 5]) => (2 4)
```

Higher order functions can be easily chained together to create complex transformations:

```
(filter even?
  (map #(* 3 %) [1 2 3 4 5]))

=>(6 12)
```

Here we multiply each item by 3, then we use `filter` to only keep the even items from the resulting sequence. Having higher-order functions means that you should rarely have to write loops or explicit recursion. When iterating a collection, use a function such as `map` or `filter` instead. Since Clojure has a rich standard library, practically any data transformation can be easily achieved by combining functions found there. See here for some examples of this approach in action.

Once you learn to associate data transformations with specific functions, many problems can be solved by simply putting these functions together in a specific order.

Let's take a look at using this idea for a simple real world problem. We'd like to display a formatted address given the fields representing it. Commonly an address has a unit number, a street, a city, a postal code, and a country. We'll have to examine each of these fields, remove the `nil` and empty ones, and insert a separator between them. Given a table containing the following fields:

```
unit        | street            | city       | postal_code | country
""          | "1 Main street"   | "Toronto"  | nil         | "Canada"
```

We would like to output the following formatted string using the strings in the table:

```
"1 Main street, Toronto, Canada"
```

All we have to do is find the functions for the tasks of removing empty fields, interposing the separator, and concatenating the result into a string:

```
(defn concat-fields [& fields]
  (clojure.string/join ", " (remove empty? fields)))

(concat-fields "" "1 Main street" "Toronto" nil "Canada")
=> "1 Main street, Toronto, Canada"
```

Notice that we didn't have to specify how to do any of the tasks when writing our code. Most of the time we simply say what we're doing by composing the functions representing the operations we wish to carry out. The resulting code also handles all the common edge cases out of the box:

```
(concat-fields) => ""
(concat-fields nil) => ""
(concat-fields "") => ""
```

In Clojure, it's common for the code to work correctly for all inputs out of the box.

Closures

We've now seen how we can declare functions, name them, and pass them as parameters to other functions. One last thing we can do is write functions that return other functions as their result. One use for such functions is to provide the functionality facilitated by constructors in object-oriented languages.

Let's say we wish to greet our guests with a warm greeting. We can write a function that will accept the greeting string as its parameter and return a function that takes the name of the guest and prints a customized greeting for that guest:

```
(defn greeting [greeting-string]
  (fn [guest]
    (println greeting-string guest)))

(let [greet (greeting "Welcome to the wonderful world of Clojure")]
  (greet "Jane")
  (greet "John"))
```

The inner function in the `greeting` has access to the `greeting-string` value since the value is defined in its outer scope. The `greeting` function is called a closure because it closes over its parameters, in our case the `greeting-string`, and makes them available to the function that it returns.

You'll also notice that we're using a form called `let` to bind the `greet` symbol and it available to any expressions inside it. The `let` form serves the same purpose as declaring variables in imperative languages.

Threading Expressions

By this point you're probably noticing that nested expressions can get difficult to read. Fortunately, Clojure provides a couple of helper forms to deal with this problem. Let's say we have a range of numbers, and we want to increment each number, interpose the number 5 between them, then sum the result. We could write the following code to do that:

```
(reduce + (interpose 5 (map inc (range 10))))
```

It's a little difficult to tell what's happening above at a glance. With a few more steps in the chain we'd be really lost. On top of that, if we wanted to rearrange any of the steps, such as interposing 5 before incrementing, then we'd have to renest all our expressions. An alternative way to write the above expression is to use the `->>` form:

```
(->> (range 10) (map inc) (interpose 5) (reduce +))
```

Here, we use `->>` to thread the operations from one to the next. This means that we implicitly pass the result of each expression as the last argument of the next expression. To pass it as the first argument we'd use the `->` form instead.

Laziness

Many Clojure algorithms use lazy evaluation where the operations aren't performed unless their result actually needs to be evaluated. Laziness is crucial for making many algorithms work efficiently. For example, you might think the preceding example is very inefficient since we have to iterate our sequence each time to create the range, map across it, interpose the numbers, and reduce the result.

However, this is not actually the case. The evaluation of each expression happens on demand. The first value in the range is generated and passed to the rest of the functions, then the next, and so on, until the sequence is exhausted. This is a similar approach that languages like Python take with their iterator mechanics.

Code Structure

One nontrivial difference between Clojure and imperative languages is the way the code is structured. In imperative style, it's a common pattern to declare a shared mutable variable and modify it by passing it different functions. Each time we access the memory location we see the result of the code that previously worked with it. For example, if we have a list of integers and we wish to square each one then print the even ones, the following Python code would be perfectly valid:

```
l = [1, 2, 3, 4, 5]

for i in l
  i = i*i

for i in l
  if (i mod 2 == 0)

print l
```

In Clojure this interaction has to be made explicit. Instead of creating a shared memory location and then having different functions access it sequentially, we chain functions together and pipe the input through them:

```
(println
  (filter #(= (mod % 2) 0)
    (map #(* % %) (range 1 6))))
```

We could also flatten out the steps using the `->>` macro introduced earlier:

```
(->> (range 1 6)
     (map #(* % %))
     (filter #(= (mod % 2) 0))
     (println))
```

Each function returns a new value instead of modifying the existing data in place. You might think that this can get very expensive, and it would be with a naÃ¯ve implementation where the entirety of the data is copied with every change.

In reality, Clojure is backed by persistent data structures that create in-memory revisions of the data. Each time a change is made a new revision is created proportional to the size of the change. With this approach we only pay the price of the difference between the old and the new structures while ensuring that any changes are localized.

Destructuring

Clojure has a powerful mechanism called destructuring for declaratively accessing values in data structures. This technique provides easy to access to the data and serves to document the parameters to a function. Let's look at some examples to see how it works.

```
(let [[smaller bigger] (split-with #(< % 5) (range 10))]
    (println smaller bigger))

=>(0 1 2 3 4) (5 6 7 8 9)
```

Above, we use `split-with` function to split a range of ten numbers into a sequence containing two elements: numbers less than 5 and numbers greater than or equal to 5. Since we know the format of the result, we can write it in a literal form as `[smaller bigger]` in the let binding. Destructuring is not limited to the `let` form and works for all types of bindings such as function arguments.

Let's look at another function called `print-user` that accepts a vector with three elements and binds them to `name`, `address`, and `phone`, respectively:

```
(defn print-user [[name address phone]]
  (println name "-" address phone))

(print-user ["John" "397 King street, Toronto" "416-936-3218"])
=> "John - 397 King street, Toronto 416-936-3218"
```

We can also specify variable arguments as a sequence. This is done by using `&` followed by the name of the list containing the remaining arguments:

```
(defn print-args [& args]
  (println args))

(print-args "a" "b" "c") => (a b c)
```

Since the variable arguments are stored in a sequence, it can be destructured like any other:

```
(defn print-args [arg1 & [arg2]]
  (println
    (if arg2
      "got two arguments"
      "got one argument")))

(print-args "bar")
=>"got one argument"

(print-args "bar" "baz")
=>"got two arguments"
```

Destructuring can also be applied to maps. When destructuring a map, we create a new map where we supply the names for the local bindings pointing to the keys from the original map:

```
(let [{foo :foo bar :bar} {:foo "foo" :bar "bar"}]
  (println foo bar)
```

It's also possible to destructure a nested data structure. As long as you know the structure of the data being passed in, you can simply write it out:

```
(let [{[a b c] :items id :id} {:id "foo" :items [1 2 3]}]
  (println id "->" a b c))
=> "foo -> 1 2 3"
```

Finally, since extracting keys from maps is a very common operation, Clojure provides syntactic sugar for this task:

```
(defn login [{:keys [user pass]}]
 (and (= user "bob") (= pass "secret")))

(login {:user "bob" :pass "secret"})
```

Another useful destructuring option allows us to extract some keys while preserving the original map:

```
(defn register [{:keys [id pass repeat-pass] :as user}]
  (cond
    (nil? id) "user id is required"
    (not= pass repeat-pass) "re-entered password doesn't match"
    :else user))
```

Namespaces

When writing real-world applications we need tools to organize our code into separate components. Object-oriented languages provide classes for this purpose. The related methods will all be defined in the same class. In Clojure, we group our functions into namespaces instead. Let's look at how a namespace is defined.

```
(ns colors)

(defn hex->rgb [[_ & rgb]]
    (map #(->> % (apply str "0x") (Long/decode))
         (partition 2 rgb)))

(defn hex-str [n]
  (-> (clojure.core/format "%2s" (Integer/toString n 16))
      (clojure.string/replace " " "0")))

(defn rgb->hex [color]
  (apply str "#" (map hex-str color)))
```

Above, we have a namespace called `colors` containing three functions called `hex->rgb`, `hex-str`, and `rgb->hex`. The functions in the same namespace can call each other directly. However, if we wanted to call these functions from a different namespace we would have to reference the `colors` namespace there first.

Clojure provides two ways to do this, we can either use the `:use` or the `:require` keywords. When we reference a namespace with `:use`, all its Vars become implicitly available as if they were defined in the namespace that references it.

```
(ns myns
  (:use colors))
```

```
(hex->rgb "#33d24f")
```

There are two downsides to this approach. We don't know where the function was originally defined, making it difficult to navigate the code, and if we reference two namespaces that use the same name for a function, we'll get an error.

We can address the first problem by selecting the functions we wish to use explicitly using the `:only` keyword in our `:use` declaration.

```
(ns myns
  (:use [colors :only [rgb->hex]]))

(defn hex-str [c]
  (println "I don't do much yet"))
```

This way we document where `rgb->hex` comes from, and we're able to declare our own `hex-str` function in the `myns` namespace without conflicts. Note that `rgb->hex` will still use the `hex-str` function defined in the `colors` namespace.

The approach of using the `:require` keyword to reference the namespace provides us with more flexible options. Let's look at each of these.

We can require a namespace without providing any further directives. In this case, any calls to Vars inside it must be prefixed with the namespace declaration indicating their origin.

```
(ns myns (:require colors))

(colors/hex->rgb "#324a9b")
```

This approach is explicit about the origin of the Vars being referenced and ensures that we won't have conflicts when referencing multiple namespaces. One problem is that when our namespace declaration is long, it gets tedious to have to type it out any time we wish to use a function declared inside it. To address this problem, the `:require` statement provides the `:as` directive, allowing us to create an alias for the namespace.

```
(ns myotherns
  (:require [colors :as c]))

(c/hex->rgb "#324a9b")
```

We can also require functions from a namespace by using the `:refer` keyword. This is synonymous with the `:use` notation we saw earlier. To require all the functions from another namespace, we can write the following:

```
(ns myns
  (:require [colors :refer :all]))
```

If we wish to select what functions to require by name, we can instead write:

```
(ns myns
  (:require [colors :refer [rgb->hex]))
```

As you can see, there's a number of options available for referencing Vars declared in other namespaces. If you're not sure what option to pick, then requiring the namespace by name or alias is the safest route.

Dynamic Variables

Clojure provides support for declaring dynamic variables that can have their value changed within a particular scope. Let's look at how this works.

```
(declare ^{:dynamic true} *foo*)

(println *foo*)
=>#<Unbound Unbound: #'bar/*foo*>
```

Here we declared `*foo*` as a dynamic Var and didn't provide any value for it. When we try to print `*foo*` we get an error indicating that this Var has not been bound to any value. Let's look at how we can assign a value to `*foo*` using a binding.

```
(defn with-foo [f]
  (binding [*foo* "I exist!"]
    (f)))

(with-foo #(println *foo*)) =>"I exist!"
```

We set `*foo*` to a string with value "I exist!" inside the `with-foo` function. When our anonymous function is called inside with-foo we no longer get an error when trying to print its value.

This technique can be useful when dealing with resources such as file streams, database connections, or scoped variables. In general, the use of dynamic variables is discouraged since they make code more opaque and difficult to reason about. However, there are legitimate uses for them, and it's worth knowing how they work.

Dynamic Polymorphism

One useful aspect of object-orientation is polymorphism, while it happens to be associated with that style it's in no way exclusive to it. Clojure provides two common ways to achieve runtime polymorphism. Let's look at each of these in turn.

Multimethods

Multimethods provide an extremely flexible dispatching mechanism using a selector function associated with one or more methods. The multimethod is defined using `defmulti` and its methods are each defined using `defmethod`. For example, if we had different shapes and we wanted to write a multimethod to calculate the area we could do the following:

```
(defmulti area :shape)

(defmethod area :circle [{:keys [r]}]
  (* Math/PI r r))

(defmethod area :rectangle [{:keys [l w]}]
  (* l w))

(defmethod area :default [shape]
  (throw (Exception. (str "unrecognized shape: " shape))))

(area {:shape :circle :r 10})
=> 314.1592653589793

(area {:shape :rectangle :l 5 :w 10})
=> 50
```

Above, the dispatch function uses a keyword to select the appropriate method to handle each type of map. This works because keywords act as functions and when passed a map will return the value associated with them. The dispatch function can be as sophisticated as we like however:

```
(defmulti encounter
  (fn [x y] [(:role x) (:role y)]))

(defmethod encounter [:manager :boss] [x y]
  :promise-unrealistic-deadlines)

(defmethod encounter [:manager :developer] [x y]
  :demand-overtime)

(defmethod encounter [:developer :developer] [x y]
  :complain-about-poor-management)

(encounter {:role :manager} {:role :boss})
=> :promise-unrealistic-deadlines
```

Protocols

Protocols allow defining an abstract set of functions that can be implemented by a concrete type. Let's look at an example protocol:

```
(defprotocol Foo
  "Foo doc string"
  (bar [this b] "bar doc string")
  (baz [this] [this b] "baz doc string"))
```

As you can see, the `Foo` protocol specifies two methods, `bar` and `baz`. The first argument to the method will be the type instance followed by its parameters. Note that the `baz` method has multiple arity. We can now create a type that implements the Foo protocol using the deftype macro:

```
(deftype Bar [data] Foo
  (bar [this param]
    (println data param))
  (baz [this]
    (println (class this)))
  (baz [this param]
    (println param)))
```

Here we create type `Bar` that implements protocol `Foo`. Each of its methods will print out some of the parameters passed to it. Let's see what it looks like when we create an instance of `Bar` and call its methods:

```
(let [b (Bar. "some data")]
  (.bar b "param")
  (.baz b)
  (.baz b "baz with param"))


some data param
Bar
baz with param
```

The first method call prints out the data `Bar` was initialized with and the parameter that was passed in. The second method call prints out the object's class, while the last method call demonstrates the other arity of baz.

We can also use protocols to extend the functionality of existing types, including existing Java classes. For example, we can use extend-protocol to extend the `java.lang.String` class with the `Foo` protocol:

```
(extend-protocol Foo String
  (bar [this param] (println this param)))

(bar "hello" "world")
=>"hello world"
```

The above examples illustrate the basic principles of how protocols can be used to write polymorphic code. However, there are many other uses for protocols as well and I encourage you to discover these on your own.

Dealing With Global State

While predominantly immutable, Clojure provides support for shared mutable data via its STM functions in the standard library. The STM is used to ensure that all updates to shared mutable variables are done atomically.

There are two primary mutable types: the `atom` and the `ref`. The `atom` is used in cases where we need to do uncoordinated updates and the `ref` is used when we might need to do multiple updates as a transaction. Let's look at an example of defining an `atom` and using it.

```
(def global-val (atom nil))
```

Above, we created an `atom` called `global-val` and its current value is `nil`. We can now read its value by using the `deref` function, which returns the current value.

```
(println (deref global-val)) => nil
```

Since this is a common operation, there is a shorthand for `deref`: the `@` symbol:

```
(println @global-val)
```

The above code is equivalent to the preceding example.

Let's look at two ways of setting a new value for our `atom`. We can either use `reset!` and pass in the new value, or we can use `swap!` and pass in a function that will be used to update the current value.

```
(reset! global-val 10) (println @global-val) =>10

(swap! global-val inc) (println @global-val) =>11
```

ï¿¼ï¿¼ Note that both `swap!` and `reset!` end in an exclamation point `!`; this is a convention to indicate that these functions modify mutable data.

We define refs the same way we define atoms, but the two are used rather differently. Let's take a quick look at how they work below.

```
(def names (ref []))

(dosync
  (ref-set names ["John"])
  (alter names #(if (not-empty %)
  (conj % "Jane") %)))
```

In this code, we define a `ref` called `names`, then open a transaction using the `dosync` statement. Inside the transaction we set `names` to a vector with the value `"John"`. Next, we call `alter` to check if `names` is not empty and add `"Jane"` to the vector of the names if that's the case.

Note that since this is happening inside a transaction, the check for emptiness depends on the existing state along with any state built up within the same transaction. If we tried to add or remove a name in a different transaction, it would have no visible effect on ours. In case of a collision, one of the transactions would end up being retried.

Writing Code That Writes Code

Clojure, being a Lisp, provides a powerful macro system. Macros allow templating repetitive blocks of code and deferring evaluation, among numerous other uses. A macro works by treating code as data instead of evaluating it. This allows us to manipulate the code tree just like any other data structure.

Macros execute before compile time and the compiler sees the result of macro execution. Because of this level of indirection, macros can be difficult to reason about, and thus it's best not to use them when a function will do the job.

Let's look at a concrete example of a macro and see how it differs from the regular code we saw previously. Imagine that we have a web application with a session atom that might contain a user. We might want to load certain content only if a user is present in the session and not otherwise.

```
(def session (atom {:user "Bob"}))
```

```
(defn load-content []
  (if (:user @session)
    "Welcome back!"
    "please log in"))
```

This will work, but it's tedious and error-prone to write out the `if` statement every single time. Since our condition's logic stays the same, we can template this function as follows:

```
(defmacro defprivate [name args & body]
  `(defn ~(symbol name) ~args
     (if (:user @session)
       (do ~@body)
       "please log in")))
```

The macros are defined using the `defmacro` special form. The major difference between `defn` and `defmacro` is that the parameters passed to `defmacro` are not evaluated by default.

To evaluate the parameter we use the `~`, as we're doing with `~(symbol name)`. Using the `~` notation indicates that we'd like to replace the name with the value it refers to. This is called unquoting.

The `~@` notation used in `(do ~@body)` is called unquote splicing. This notation is used when we're dealing with a sequence. The contents of the sequence will be merged into the outer form during the splicing. In this case the body consists of a list representing the function's body. The body must be wrapped in a `do` block because the `if` statement requires having no more than three arguments.

The ` sign means that we wish to treat the following list as data instead of executing it. This is the opposite of unquoting, and it's referred to as syntax-quoting.

As I mentioned earlier, the macros are executed before compile time. To see what the macro will be rewritten as when the compiler sees it, we can call `macroexpand-1`.

```
(macroexpand-1 '(defprivate foo [greeting] (println greeting)))

(clojure.core/defn foo [greeting]
  (if (:user (clojure.core/deref session))
    (do (println greeting))
    "please log in"))
```

We can see that `(defprivate foo (println "bar"))` gets rewritten with a function definition that has the `if` statement inside. This resulting code is what the compiler will see, and it's equivalent to what we would have to write by hand otherwise. Now we can simply define a private function using our macro, and it will do the check for us automatically.

```
(defprivate foo [message] (println message))

(foo "this message is private")
```

The preceding example might seem a little contrived, but it demonstrates the power of being able to easily template repetitions in code. This allows creating a notation that expresses your problem domain using the language that is natural to it.

The Read-Evaluate-Print Loop

Another big aspect of working in Clojure is the read-evaluate-print loop (REPL). In many languages you write the code, then run the entire program to see what it does. In Clojure, most development is done interactively using the REPL. In this mode we can see each piece of code we write in action as soon as it's written.

In nontrivial applications it's often necessary to build up a particular state before you can add more functionality. For example, a user has to log in and query some data from the database, then you need to write functions to format and display this data. With a REPL you can get the application to the state where the data is loaded and then write the display logic interactively without having to reload the application and build up the state every time you make a change.

This method of development is particularly satisfying because you see immediate feedback when making changes. You can easily try things out and see what approach works best for the problem you're solving. This encourages experimentation and refactoring code as you go, which in turn helps you to write better and cleaner code.

Calling Out to Java

One last thing that we'll cover is how Clojure embraces its host platform to benefit from the rich ecosystem of existing Java libraries. In some cases we may wish to call a Java library to accomplish a particular task that doesn't have a native Clojure implementation. Calling Java classes is very simple, and follows the standard Clojure syntax fairly closely.

Importing Classes

When we wish to use a Clojure namespace, we employ either the `:use` or the `:require` statements discussed above. However, when we wish to import a Java class, we have to use the `:import` statement instead:

```
(ns myns
  (:import java.io.File))
```

We can also group multiple classes from the same package in a single import, as follows:

```
(ns myns
  (:import [java.io File FileInputStream FileOutputStream]))
```

Instantiating Classes

To create an instance of a class, we can call new just as we would in Java:

```
(new File ".")
```

There is also a commonly used shorthand for instantiating objects:

```
(File. ".")
```

Calling Methods

Once we have an instance of a class, we can start calling methods on it. The notation is similar to making a regular function call. When we call a method, we pass the object its first parameter followed by any other parameters that the method accepts.

```
(let [f (File. ".")]
  (println (.getAbsolutePath f)))
```

Above, we created a new file object `f`, and then called the `.getAbsolutePath` method on it. Notice that methods have a period `.` in front of them to differentiate them from a regular Clojure function. If we wanted to reference a static method or a variable in a class, we would use the `/` notation instead:

```
(str File/separator "foo" File/separator "bar")

(Math/sqrt 256)
```

There's also a shorthand for chaining multiple method calls together using the `..` notation. Say we wanted to get the string indicating the file path and then get its bytes; we could write the code for that in two ways.

```
(.getBytes (.getAbsolutePath (File. ".")))

(.. (File. ".") getAbsolutePath getBytes)
```

Further Reading

This concludes our tour of Clojure basics. While we only touched on only a small portion of the overall language, I hope that the guide has provided you with a bit of insight into how idiomatic Clojure code is written. Below are some useful links for more in-depth documentation about the language.

- Official Clojure documentation
- Community Clojure documentation site
- A humorous introduction to Clojure
- Clojure API documentation
- Clojure cheatsheet
- Clojure style guide