An In-Depth Look at Clojure Collections

Posted by **Michael Fogus and Chris Houser** on May 06, 2010

Sections
- Architecture & Design,
- Development

Topics
- Clojure ,
- Java ,
- Languages ,
- Programming

InfoQ readers can get an exclusive 35% discount on any version of The Joy of Clojure (Early Access Ebook, or Early Access Ebook+Print book) with the checkout code "infoq35". The offer is valid until June 16, 2010.

If you're familiar with the Clojure programming language, then you might know that at its heart lays a powerful set of immutable, persistent, collection types.In this article we will talk a bit about the underpinnings of these collection types including a deep dive into a couple of them; namely its vectors and maps. Finally, we'll wrap up by presenting an example of how viewing a problem through the lens of the Clojure way we can vastly simply our design.

On Immutability

Why was Clojure designed and implemented with immutability as a cornerstone principle? While certainly no panacea, fostering immutability at the language level solves many difficult problems right out of the box, while at the same time simplifying many others. Coming from a language background where mutability interwoven with imperative programming methods reign, it often requires a significant conceptual leap to twist one's mind to accept and utilize immutability and functional

programming. In many cases, purists see the essence of immutability as a precondition for any given language to fall into the category of functional programming. While we are certainly not willing to set hard and fast definitions of such nebulous topics, we will say that data structure immutability and functional programming do indeed complement one another quite well. In this section we'll build a conceptual basis for immutability as it relates to Clojure's underlying philosophy as well as why you should work to foster immutability even when outside the warming confines of Clojure proper.

Defining Immutability

When using the word immutability in the context of Clojure, we will more often than not refer strictly to its core data types. However, the larger picture of immutability can be extended beyond the strict ecosystem of data structures alone and instead encompass any object that adheres to the notion of immutability. In many cases, when talking specifically about Clojure's immutable data structures we could in fact be talking about the broader category of immutable objects without loss of meaning. However, in order to make this case we should probably set down some conditions defining just what we mean by immutability.

Every Day is Like Sunday

There is an entire branch of philosophy named predestination devoted to exploring the notion that there is no such thing as freewill, but instead, everything that we are or ever will be is determined beforehand. While this possibility for our own lives may seem bleak, the very notion does nicely encapsulate the first principle of object immutability. That is, all of the possible properties of immutable objects are defined at the time of their construction and can not be changed thereafter.

Immutability Exists Through Convention

There is nothing magical about immutable objects. That is, computer systems are in many ways open systems, providing the keys to the vault if one is so inclined to grab them. However, in order to foster an air of immutability in our own systems it is of the utmost importance to crate a facade of immutability. In other words, immutability requires that we layer over and abstract the parts of our system that provide unrestrained mutability. For example, creating immutable classes in Java[1] requires us to do this in a number of ways. First, a class itself and all of its fields should be labeled as final. Next, in no way should an objects's this reference escape during construction. And finally, any internal mutable objects should originate, either whole-cloth or through a copy, within the class itself and thus never escape. Obviously we're simplifying in order to illustrate as there are finer details to this recipe for Java immutability, but for now these simplified highlights serve to show that by observing convention, even an inherently mutable language such as Java can be made to be immutable. In languages like Java, immutability is not directly supported and requires some gymnastics in order to achieve while Clojure directly supports immutability as a language feature[2] with its core data structures. By providing immutable data structures as a primary language feature Clojure separates[3] the complexity of working with immutable structures from the complexities of their implementation.

Of course the parts defining immutable objects as outlined above are meant to be viewed agnostic to any specific programming language. However, by providing immutability either as a core language feature or through convention, you can buy yourself enormous benefits.

The Benefits of Immutability

Clojure's immutable data structures are not bolted on to the language as an after-thought or as a choice in an ala-carte menu. Instead, their inclusion in the language runs deep to its philosophical core.

Invariants

Invariant-based programming involves the definition of constraints on classes and function in order to provide assurances that if its instances enter into certain states assertion errors will arise. Providing invariants within a mutable system requires a fair amount of assertion weaving within the methods of any given class. However, by observing a practice of immutability, invariants are defined solely within the construction mechanism and can never be violated thereafter.

Reasoning

In the face of mutability, passing any object to any function works to obliterates the possibility of tracking the changes in its properties. However, because the life of an immutable object is one of pre-destiny, the matter of reasoning about its possible states becomes a trivial matter. It therefore follows that the act of testing such a system is simplified in that set of the possible states and transitions is constrained.

Equality Has Meaning

Equality in the presence of mutability has no meaning. Equality in the face of mutability and concurrency is utter lunacy. That is, if any two objects resolve as being equal now, then there is absolutely no guarantee that they will a moment from now. And if two objects are not equal forever, then they are technically never equal[4]. Providing immutable objects once again assigns meaning to equality in that if two object are equal now then they will always be so.

Sharing is Cheap

If you are certain that an object will never change, then sharing said object becomes a simple matter of providing a reference to it. In Java to do so often requires a lot of defensive copying. Along this vein, because we can freely share references for immutable objects we can likewise intern them for free.

Flattening the Levels of Indirection

There is a marked difference between a mutable object and a mutable reference. The default in Java is that there are mutable references that contain mutable data. However, in Clojure there are only mutable references. This may seem like a minor detail, but it certainly works to reduce unnecessary complexities. Why else is there a proliferation of final in next-generation Java programming styles?

Of course, in a concurrency-oriented programming language like Clojure, the primary benefit of immutability is that the core types can be shared freely among separate threads without fear. In the next section we will discuss this particular benefit in more detail.

Immutability Fosters Concurrent Programming

Immutable objects are always thread safe -- Brian Goetz, Java Concurrency in Practice

If an object cannot change it follows that it can be shared freely between different threads of execution without fear of concurrent modification. There can be very little debate about this particular point, but that fact does not answer the question of just how mutation occurs. Without delving into the specifics, you likely already know that Clojure isolates mutation to its reference types while the data wrapped with them is left unchanged. It's enough to leave this well enough alone for now since we will devote chapter 9 to this and related topics.

Designing a Persistent Toy

We will not go into terrible detail about the internals of Clojure's persistent data structures -- we'll leave that to others[5]. But we do want to explore the notion of structural sharing. Our example will be highly simplified compared to Clojure's implementations, but it should help clarify some of the techniques used.

The simplest shared-structure type is the list. Two different items can be added to the front of the same list, producing two new lists which share their next parts. Lets try this out by actually creating a base list and then two new lists from that same base:

```
(def baselist (list :barnabas :adam))
(def lst1 (cons :willie baselist))
(def lst2 (cons :phoenix baselist))


lst1
;=> (:willie :barnabas :adam)

lst2
;=> (:phoenix :barnabas :adam)
```

We can think of baselist as a historical version of both lst1 and lst2. But it's also the shared part of both lists. More than being equal, the next parts of both lists are *identical* -- the very same instance:

```
(= (next lst1) (next lst2))
;=>gt; true


(identical? (next lst1) (next lst2))
;=>gt; true
```

So that's not too complicated, right? But the features supported by lists are also pretty limited. Clojure's vectors and maps also provide structural sharing, while allowing you to change values anywhere in the collection, not just on one end. The key is the tree structure each of these uses internally. To help demonstrate how a tree can allow interior changes and maintain shared structure at the same time, let's build one of our own.

Each node of our tree will have 3 fields: a value, a left branch, and a right branch. We'll just put them in a map, like this:

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

That's the simplest possible tree -- a single node holding the value 5, with empty left and right branches. This is exactly the kind of tree we want to return when a single item is added to an empty tree. To represent an empty tree, we'll just use nil. Let's write our own conj function to build up our tree, starting with just the code for this initial case:

```
(defn xconj [t v]
  (cond
    (nil? t) {:val v, :l nil, :r nil}))


(xconj nil 5)
{:val 5, :l nil, :r nil}
```

Hey, it works! Not too impressive yet though, so we need to handle the case where an item is being added to a non-empty tree. Lets keep our tree in order by putting values less than a node's :val in the left branch, and other values in the right branch. That means we need a test like:

```
(< v (:val t))
```

When that's true, we need our new value v to go into the left branch, (:l t). If this were a mutable tree, we'd *change* the value of :l to be our new node. Instead, let's build a *new* node, copying in the parts of the old node that don't need to change. Something like:

```
{:val (:val t),
 :l (insert-new-val-here),
 :r (:r t)}
```

This will be our new root node. Now we just need to figure out what to put for insert-new-val-here. If the old value of :l is nil, we simply need a new single-node tree -- we even have code for that already, so could use (xconj nil v). But what if :l is not nil? In that case we just want to insert v in its proper place within whatever tree :l is pointing to, so (:l t) instead of nil in that xconj expression. That gives us a new xconj that looks like this:

```
(defn xconj [t v]
  (cond
    (nil? t)       {:val v, :l nil, :r nil}
    (
```

There, it's working. At least it seems to be -- there's a lot of noise in that output making it difficult to read. Here's a little function to traverse the tree in sorted order, converting it to a seq that will print more succinctly:

```
(defn xseq [t]
  (when t
    (concat (xseq (:l t)) [(:val t)] (xseq (:r t)))))

(xseq tree1)
;=>gt; (2 3 5)
```

Lookin' good. Now we just need a final condition for handling the insertion of values that are *not* less than the node value:

```
(defn xconj [t v]
  (cond
    (nil? t)       {:val v, :l nil, :r nil}
    (
```

Now that we've got the thing built, hopefully you understand well enough how it's put together that this demonstration of the shared structure will be unsurprising:

```
(def tree2 (xconj tree1 7))
(xseq tree2)
```

```
;=>gt; (2 3 5 7)

(identical? (:l tree1) (:l tree2))
;=>gt; true
```

Think about that for a moment. No matter how big the left side of a tree's root node is, something can be inserted on the right side without copying, changing, or even examining the left side. All those values will be included in the new tree, along with the inserted value. Thus this toy example demonstrates several features that it has in common with all of Clojure's persistent collections:

- Every "change" creates at least a new root node, plus new nodes as needed in the path through the tree to where the new value is being inserted.
- Values and unchanged branches are never copied, but references to them are copied from nodes in the old tree to nodes in the new one.
- This implementation is completely thread-safe in a way that's easy to check -- no object that existed before a call to xconj is changed in any way, and newly created nodes are in their final state before being returned. There is simply no way for any other thread, or even any other functions in the same thread, to see anything in an inconsistent state.

There are a few ways our example falls down though, when compared to Clojure's rather more production-quality code. This toy:

- Is just a binary tree [6]
- Can only store numbers
- Will overflow the stack if the tree gets too deep
- Produces (via xseq) a non-lazy seq that will contain a whole copy of the tree.
- Can create unbalanced trees that will have very bad "worst case" algorithmic complexity [7].

While structural sharing as described using xconj as a basis example can reduce the memory footprint of persistent data structures, it alone is insufficient. Instead, Clojure leans heavily on the notion of lazy sequences to further reduce its memory footprint as we explore further in our book, *The Joy of Clojure*. Having touch on the basics of structural sharing that is analogous to Clojure's collection types, let's spend some time talking about the two most ubiquitous collection types in Clojure: vectors and maps.

Vectors: Creating and Using All Their Varieties

A vector stores zero or more values sequentially indexed by number, a bit like an array, but are of course immutable and persistent. They are versatile and make efficient use of memory and processor resources at both small and large sizes.

Vectors are probably the most frequently used collection type in Clojure code. They're used as literals for argument lists and let bindings, for holding large amounts of application data, as stacks, and as map entries. We'll examine all of these facets, and finish with an examination of cases where vectors do not work well.

Building Vectors

Vectors can be built using the literal square-bracket syntax [1 2 3]. This syntax, compared to lists' round parenthesis, is one reason you might choose to use a vector over a list. For example, the let form would work perfectly well, and with a nearly identical implementation, if it took a literal *list* of bindings instead of a literal *vector*. However the square brackets are visually different from the round

parenthesis surrounding the let form itself as well as the likely function calls in the body of the let form, and this is quite useful for us humans. Using vectors to indicate bindings for let, with-open, fn and such is idiomatic in Clojure and is a pattern you are encouraged to follow in any similar macros you write yourself.

The most common way to create a vector is with the literal syntax described above. But in many cases you want to create a vector out of the contents of some other kind of collection. For this there is the function vec:

```
(range 10)
;=>gt; (0 1 2 3 4 5 6 7 8 9)

(vec (range 10))
;=>gt; [0 1 2 3 4 5 6 7 8 9]
```

Note the round parens on the result from range, compared to the square brackets on the result from vec. This is our hint that range returned a seq, while vec returned a vector filled with the values from the seq.

If you already have a vector but want to concatenate several values to it, into is your friend:

```
(let [my-vector [:a :b :c]]
  (into my-vector (range 10)))

;=> [:a :b :c 0 1 2 3 4 5 6 7 8 9]
```

If you want it to return a vector, the first argument to into *must* be a vector, as we have above. The second arg however can be any sequence, such as what range returns, or anything else that works with seq function. The values from the seq are "poured into" the supplied vector.

Because vectors themselves work with seq, you can use into to concatenating two vectors together:

```
(into [:a :b :c] [:x :y :z])
;=>gt; [:a :b :c :x :y :z]
```

But be aware this is treating the second vector just as a seq, so it's an essentially linear-time operation based on the size of the second vector[8].

That's really all you need to know about building vectors, though just for completeness we'll show you the rarely-used vector function, which builds a vector from of the arguments you give it instead of from a seq:

```
(vector 1 2 3)
;=> [1 2 3]
```

There's hardly ever a good reason to use vector instead of a literal vector. This is more true of vectors than it is of lists, but that's getting ahead of ourselves.

Using vec and into, it's easy to build vectors much larger than are usually built using vector literals. Once you've got a large vector like that, what are you doing to do with it? Let's look at that next.

Large Vectors

When collections are small, the performance differences between vectors and lists hardly matters at all. But as they get larger, each becomes dramatically slower at operations the other can still perform efficiently. Vectors are particularly efficient at three things relative to lists: adding or removing things

from the right-hand end of the collection, accessing or changing items in the interior of the collection by numeric index, and walking in reverse order. Adding and removing from the end is done by treating the vector as a stack -- we'll cover that a bit later.

Any item in a vector can be accessed by its index number from 0 up to but not including (count my-vector) in essentially constant time[9]. This can be done using the function nth; the function get, essentially treating the vector like a map; or by simply invoking the vector itself as a function. Let's look at each of these as applied to this example vector:

```
(def some-chars (vec (map char (range 65 75))))

some-chars
;=>gt; [\A \B \C \D \E \F \G \H \I \J]
```

All three of these do the same work and each returns \E:

```
(nth some-chars 4)
(get some-chars 4)
(some-chars 4) ; vector as a function
```

Which to use is a judgement call, so here are some things you might consider when choosing:

	nth	get	vector as function
Vector is nil	returns nil	returns nil	throws exception
Index out of range	throws exception	returns nil	throws exception
Supports Ònot foundÓ	yes	yes	no

Because vectors are indexed, they can be efficiently walked in either direction, left-to-right or right-to-left. The seq and rseq functions return sequences that do exactly that:

```
(seq some-chars)
;=>gt; (\A \B \C \D \E \F \G \H \I \J)

(rseq some-chars)
;=>gt; (\J \I \H \G \F \E \D \C \B \A)
```

Any item in a vector can be "changed" using the assoc function. Of course since vectors are immutable it isn't actually changed -- assoc returns a new vector that is exactly like the old one except for the updated value. Clojure does this in essentially constant time using structural sharing between the old and new vectors as we described at the beginning of this article.

```
(assoc some-chars 4 "no longer E")
;=> [\A \B \C \D "no longer E" \F \G \H \I \J]
```

Note that the value of some-chars hasn't changed at all. The REPL just printed a new vector that is almost but not quite entirely like some-chars.

The assoc function for vectors only works on indices that already exist in the vector or, as a special case, exactly one step past the end. In this latter case, the returned vector will be one item larger than the input vector. More frequently however vectors are "grown" using the conj function as we'll see in

the next section.

There are a few higher-powered functions provided that use assoc internally. For example, the replace function works on both seqs and vectors, but when given a vector it uses assoc to fix up and return a new vector:

```
(replace {2 :a, 4 :b} [1 2 3 2 3 4])
;=> [1 :a 3 :a 3 :b]
```

The functions assoc-in and update-in are for working with nested structures of vectors and/or maps, like this one[10]:

(def matrix

```
(def matrix
     [[1 2 3]
      [3 4 5]
      [5 6 7]])
```

Both assoc-in and update-in take a series of indices to pick items from each more deeply nested level. For a vector arranged like the matrix example above, this amounts to row and column coordinates:

```
(assoc-in matrix [1 2] 'x)
;=> [[1 2 3] [3 4 x] [5 6 7]]
```

The update-in function works the same way, but instead of taking a value to *overwrite* an exiting value, it takes a function to *apply* to an existing value. It will replace the value at the given coordinates with the return value of the function you give:

```
(update-in matrix [1 2] * 10)
;=> [[1 2 3] [3 4 50] [5 6 7]]
```

The coordinates refer to the value 5, and the function given above is * so the value 5 will be replaced by the return value of (* 5 10). The trailing 10 there comes from the extra argument given to update-in -- any number of arguments may be given to be appended to the function applied.

So far we've not changed the size of any of the vectors we've looked at, just looked things up and changed items in place. Next we'll look at growing and shrinking vectors -- treating them like stacks.

Vectors as Stacks

Classic mutable stacks have at least two operations, *push* and *pop*. These also work work on vectors, though they're called conj and pop respectively, by adding and removing things from the right-hand end. Because vectors are immutable, pop returns a new vector with the right-most item dropped -- this is a bit different from the many mutable stack APIs which generally return the dropped item. Consequently, peek becomes more important as the primary way to get item from the top of the stack.

Enough talking -- let's do some doing.

```
(def my-stack [1 2 3])

(peek [1 2 3])
;=>gt; 3

(pop [1 2 3])
;=>gt; [1 2]
```

```
(conj my-stack 4)
;=>gt; [1 2 3 4]

(->gt; my-stack (conj 4) (conj 5) pop (conj 6))
;=>gt; [1 2 3 4 6]
```

Each of these operations completes in essentially constant time. Most of the time, a vector that's used as a stack is used that way throughout its life. It's helpful to future readers of your code to keep this is mind and use the stack operations consistently, even when other functions might work. For example, last on a vector returns the same thing as peek, but besides being slower it leads to unnecessary confusion about how the collection is being used. If the algorithm involved calls for a stack, use conj not assoc for growing the vector, peek not last, and pop not dissoc for shrinking it.

If conj and pop aren't fast enough for you, faster equivalents are available for vectors that support transient which we will cover in our book.

The functions conj, pop, and peek work on any object that implements clojure.lang.IPersistentStack[11]. Besides vectors, Clojure lists also implement this interface, but the functions operate on the left-hand side of lists instead of the right-hand side as with vectors.

Using Vectors instead of 'reverse'

The ability of vectors to grow efficiently on the right-hand end, and then be walked left-to-right produces a noteworthy emergent behavior: idiomatic Clojure code very rarely uses the reverse function. This is quite different from most Lisps and Schemes, so if you're familiar with one of those, we can take a short look at how this works out. If you don't care about Lisps other than Clojure (and why would you, really), feel free to skip on down to "Vectors as Stacks".

When processing a list, it is pretty common to want to produce a new list in the same order. However if all you've got are classic Lisp lists, often the most natural algorithm[12] leaves you with a backwards list that needs to be reversed. Here's an example of a function a bit like Clojure's map, but strict instead of lazy:

```
(defn strict-map1 [f coll]
  (loop [coll coll, acc nil]
    (if (empty? coll)
      (reverse acc)
      (recur (next coll) (cons (f (first coll)) acc)))))

(strict-map1 odd? (range 5))
;=>gt; (false true false true false)
```

This is perfectly good, idiomatic Clojure code, except for that glaring reverse of the final return value. After the entire list has been walked once to produce the desired values, reverse walks it again to get them in the right order. This is both inefficient and non-idiomatic. One way to get rid of the reverse is to use a vector instead of a list as the accumulator:

```
(defn strict-map2 [f coll]
  (loop [coll coll, acc []]
    (if (empty? coll)
      acc
      (recur (next coll) (conj acc (f (first coll)))))))

(strict-map2 odd? (range 5))
;=>gt; [false true false true false]
```

A small change, but the code is now a touch cleaner and a bit faster. It does return a vector instead of a list, but this is rarely a problem as any client code that wants to treat this as a seq can usually do so automatically. On occasion, though, it may be appropriate to return (seq acc) instead of just acc.

By the way, another way to get rid of a reverse is to build a lazy sequence instead of a strict collection. In fact, this is how Clojure's own map function is implemented. We cover that and much more about lazy seqs in chapter 5 of our book.

The term "vector" in Clojure is actually a colloquial description of any object that implements clojure.lang.IPersistentVector. There are a few concrete classes that act as vectors: PersistentVector, SubVector, and MapEntry[13] So far the examples we've seen have all been plain PersistentVectors, though they would have worked for any of these types. But lets turn now to the special features of these other vector types, starting with SubVector.

SubVectors

Although items cannot be removed efficiently from a vector (except the right-most item), SubVectors provide a very fast way to take a slice of an existing vector based on start and end indexes. They are created using the subvec function.

Let's use the some-chars vector we defined earlier:

```
some-chars
;=>gt; [\A \B \C \D \E \F \G \H \I \J]
```

The subvec takes a start and end index and returns a new vector:

```
(subvec some-chars 3 6)
;=>gt; [\D \E \F]
```

The first index given to subvec is inclusive (starts *at* index 3) but the second is exclusive (ends *before* index 6).

But the really interesting thing is that this new vector internally hangs onto the entire original some-chars vector. Each lookup we do on the new vector causes the SubVector to do a little offset math and then lookup in the original vector. This makes creating a SubVector very fast. If either the SubVector or its underlying vector were mutable, you could detect this fact by making a change to one and observing the effect in the other. But since both are immutable, SubVector can be treated exactly the same as any other vector.

You can use subvec on any kind of vector and it'll work fine. But it has special logic for taking a subvec of a subvec, in which case the newest SubVector keeps a reference to the *original* vector, not the intermediate SubVector. This prevents SubVectors-of-SubVectors from stacking up needlessly, and keeps both the creation and use of the sub-subvecs fast.

Vectors as MapEntries

Clojure's hash-map, just like hash tables or dictionaries in many other languages, has a mechanism to iterate through the entire collection. Clojure's solution for this iterator is, unsurprisingly, a seq. Each item of this seq needs to include both the key and the value, so they're wrapped in a MapEntry. When printed, each entry looks like a vector:

```
(first {:width 10, :height 20, :depth 15})
;=>gt; [:width 10]
```

But not only does a MapEntry look like a vector, it really is one:

```
(vector? (first {:width 10, :height 20, :depth 15}))
;=> true
```

This means we can use all the regular vector functions on it: conj, get, and so on. It even supports destructuring, which can be very handy. For example, the locals dimension and amount below will take on the value of each key/value pair in turn:

```
(doseq [[dimension amount] {:width 10, :height 20, :depth 15}]
  (println (str (name dimension) ":") amount "inches"))
; width: 10 inches
; height: 20 inches
; depth: 15 inches
;=>gt; nil
```

Two-element vectors, whether they are MapEntries or not, can also be used to add values to a hash-map or sorted-map, if you prefer to use conj instead of assoc:

```
(conj {} [:a 1] [:b 2])
;=> {:b 2, :a 1}
```

But a MapEntry is its own type and supports a couple extra functions for retrieving its contents: key and val, which do exactly the same thing as (nth my-map 0) and (nth my-map 1), respectively. These are sometimes useful for the clarity they can bring to your code, but quite frequently destructuring is used instead as it's just so darned handy.

So now you know what vectors are, what specific kinds of vectors are included in Clojure, and some of the things that they're good at doing. To round out our understanding of vectors, let's conclude with a brief look at things that vectors are *bad* at doing.

What Vectors Are Not

Vectors are quite versatile, but there are some commonly desired patterns where they might *seem* like a good solution but in fact are not. While we prefer to focus on the positive, hopefully a few negative examples will help you escape from using the wrong tool for the job.

Vectors are not sparse

If you have a vector of length *n*, the only position where you can insert a value is at index *n*, that is appending to the far right-hand end. You can't skip some indexes and insert at a higher index number. If you want a collection indexed by non-sequential numbers, consider a hash-map or sorted-map.

Although you can replace values within a vector, you can't insert or delete items such that indexes for the subsequent items would have to be adjusted. Clojure doesn't currently have a native persistent collection that supports this kind of operation, but finger trees may help for some use cases.

If you always will be wanting to prepend things at the left-hand end of a sequential collection, instead of the right-hand end the way vectors do, you might have found a good opportunity to use a PersistentList.

Vectors are not queues

Some people have tried to use vectors as queues. One approach would be to push onto the right-hand

end of the vector using conj and then to pop items off the left using rest or next. The problem with this is that rest and next returns seqs, not vectors, so subsequent conj operations would not behave as desired. Using into to convert the seq back into a vector is O(n) which is less than ideal for every "pop".

Another approach is to use subvec as a "pop", leaving off the left-most item. Since subvec does return a vector, subsequent conj operations will push onto the right-hand end as desired. However, as we described above, subseq maintains a reference to the entire underlying vector, so none of the items being "popped" this way will ever be garbage collected. Also less than ideal.

So what *would* be the ideal way to do queue operations on a persistent collection? Why, use a PersistentQueue, of course, which Clojure provides.

Vectors cannot contain primitives

Java primitives like int and byte take less memory and are often much faster to process than their boxed counterparts Integer and Byte. Unfortunately, none of Clojure's persistent collections support primitives yet. Any attempt to put a primitive into a vector will result in auto-boxing of the item, and any value retrieved from a vector will similarly be an Object. Although there are plans for Clojure vectors to eventually support primitives, that's a ways off yet. If you really need the memory or performance benefits of primitives in a collection, for now your only choice is to use a Java array.

Vectors are not sets

If you want to find out if a vector contains a particular value, you might be tempted to use the contains? function, but you'd be disappointed by the results. contains? is for asking whether a particular *key*, not *value*, is in a collection, which is rarely useful for a vector. Vector objects do have a contains method that you can call via Java interop that will give the results you want, but Clojure offers something much better -- a real set collection. See section 6 of this chapter for details on sets.

Well there, that was certainly enough detail about vectors. We saw how to create them using literal syntax or by building them up programmatically. We looked at how to push them, pop them, slice them, and stuff them into maps. We also looked at some of the things vectors can't do well. One of these was adding and removing items from the left-hand side; though vectors can't do this, lists can. Let's look at those next.

Map Varieties and How to Use Them

Compared to the other composite data types Clojure provides, it seems that its map should be unambiguous in its use cases. While this is certainly true in many cases, there are some points to be made concerning the differing flavors of map provided. Additionally, we will make a small point about using maps in many instances where you might be tempted to use other structures. However, in general a simple rule applies: maps should be used in any case where your data structure requires a logical mapping of keys to values.

It's very difficult to write a program of any significant size without the need for a map of some sort. The use of maps are ubiquitous in writing software because frankly it's difficult to imagine a more robust data structure. However, we as programmers tend to view maps as a special case structure outside of the normal exaltations bestowed upon classes. That is, the object-oriented school of thought has relegated the map as a supporting player in favor of the class. We're not going to talk about the

merits, or lack thereof, for this relegation here, but throughout our book we discuss moving away from thinking in classes and instead thinking in sequence abstractions, maps, sets, and types. Having said all of that, it need hardly be mentioned that maps should be used to store named values. However, there are times when you may wish to use positional elements within a sequence to represent some structure. This approach tends to be brittle and can often be better served by using maps instead because they are self organizing through the use of named values. Clojure's maps support an arbitrarily complex nesting of persistent types for both keys and values, so if you want a map that has vectors of maps as its keys, that's perfectly okay. Let's talk about each form of map that Clojure provides and the tradeoffs surrounding each.

Hash Maps

Arguably, the most ubiquitous[14] form of map found in Clojure programs is the hash map which provides an unsorted key-value associative structure. In addition to the literal syntax touched on in chapter 1, hash maps can be created using the hash-map function which likewise takes alternating key/value pairs, with or without commas:

```
(hash-map :a 1, :b 2, :c 3, :d 4, :e 5)
```

The difference in results between the literal form and construction function is that the latter will always provide a hash map while the former could at times provide an array map. The reason for this incongruity is not important at the moment, but will be discussed in the upcoming section about array maps.

Clojure hash maps support heterogeneous keys, meaning that they can be of any type and each key can be of a differing type.

```
(let [m {:a 1, 1 :b, [1 2 3] "4 5 6"}]
  [(get m :a) (get m [1 2 3])])

;=>gt; [1 "4 5 6"]
```

As we previously mentioned in the beginning of this chapter, many of Clojure's composite types can be used as functions and in the case of maps they are functions of their keys. Using maps in this way will act the same as the use of the get function in the previous code sample.

```
(let [m {:a 1, 1 :b, [1 2 3] "4 5 6"}]
  [(m :a) (m [1 2 3])])

;=>gt; [1 "4 5 6"]
```

Providing a map to the seq function will return a sequence of map entries:

```
(seq {:a 1, :b 2})
;=>gt; ([:a 1] [:b 2])
```

Of course, the sequence above appears to be composed of the sets of key/value pairs contained in vectors and for all practical purposes should be treated as such. In fact, a new hash map can be created idiomatically using this precise structure:

```
(into {} (list [:a 1] [:b 2]))
;=>gt; {:a 1, :b 2}
```

Even if your embedded pairs are not vectors they can be made to be for building a new map:

```
(into {} (map vec '((:a 1) (:b 2))))
;=>gt; {:a 1, :b 2}
```

In fact, your pairs do not have to be explicitly grouped as you can use apply to create a hash map given that the key/value pairs are laid out in a sequence consecutively.

```
(apply hash-map [:a 1 :b 2])
;=>gt; {:a 1, :b 2}
```

You can also use apply in this way with sorted-map and array-map. Another idiomatic way to build a map is to use zipmap to "zip" together two sequences, the first of which contains the desired keys and the second their corresponding values:

```
(zipmap [:a :b] '(1 2))
;=>gt; {:b 2, :a 1}
```

Our use of zipmap illustrates nicely the final property of hash maps. That is, hash maps in Clojure have no order guarantees. If you do require ordering, then you should use sorted maps, discussed next.

Keeping Your Keys in Order With Sorted Maps

There is one potential downside to using maps in place of other common structures such as lists and vectors; order agnosticism. That is, it's impossible rely on a specific ordering of the key values pairs for a standard Clojure map because there are no order guarantees at all. Should the ordering of a map's entries be important for your applications, then Clojure has two map implementations providing order assurances. The first of these, which we will talk about is the sorted map which can be constructed using one of a pair of functions: sorted-map and sorted-map-by.

By default, the function sorted-map will build a map sorted by the comparison of its keys:

```
(sorted-map :thx 1138 :r2d 2)
;=>gt; {:r2d 2, :thx 1138}
```

However, you may require an alternative key ordering or perhaps for keys not normally comparable, say for non-trivial keys. In these cases you must use sorted-map-by which takes an additional comparison function:

```
function:
(sorted-map "bac" 2 "abc" 9)
;=>gt; {"abc" 9, "bac" 2}

(sorted-map-by #(compare (subs %1 1) (subs %2 1)) "bac" 2 "abc" 9)
;=>gt; {"bac" 2, "abc" 9}
```

This means that sorted maps do not generally support heterogeneous keys the way hash-maps do, although it depends on the comparison[15] function provided, for example the one above assumes all keys are strings. The default sorted-map comparison function supports maps whose keys are all Numbers or are Comparable with each other. Attempts to use keys that aren't supported by whichever comparison function you're using will generally result in a class cast exception:

```
(sorted-map :a 1, "b" 2)
;=>gt; java.lang.ClassCastException: clojure.lang.Keyword cannot be cast to
java.lang.String
```

One unique feature supported by sorted maps (and also sorted sets) is the ability to jump efficiently to a

particular key and walk forward or backward from there through the collection. This is done with the subseq and rsubseq functions for forward and backward respectively. Even if you don't know the exact key you want, these functions can be used to "round up" the next closest key that exists.

Another way that sorted maps and hash maps differ is in their handling of numeric keys. A number of a given magnitude can be represented by many different types, for example 42 can be a long, int, float, etc. Hash maps would treat each of these different objects as *different*, while a sorted map would treat them as the same. You can see the contrast in this example, where the hash map keeps both keys while the sorted map keeps just one:

```
(assoc {1 :int} 1.0 :float)
;=>gt; {1.0 :float, 1 :int}

(assoc (sorted-map 1 :int) 1.0 :float)
;=>gt; {1 :float}
```

Sorted maps will otherwise work just like a hash map and can be used interchangeably. You should use sorted maps if you need to specify or guarantee a specific key ordering. On the other hand, if you need to maintain insertion ordering, then the use of array maps are required as we will see currently.

Keeping Your Insertions in Order With Array Maps

When dealing with map literals in Clojure you might be surprised by the following:

```
{:x 1 :x 2 :x 3}
;=> {:x 1, :x 2, :x 3}

(:x {:x 1 :x 2 :x 3})
;=> 1
```

It seems counter to logic that a map could allow duplicate keys. While this is certainly true, this is a side-effect of Clojure's current implementation of map literals. That is, in order to build map literals as quickly as possible Clojure uses an array map to store its instances. An array map in turn stores its key value pairs as alternating elements within a flat array which can be populated very quickly by ignoring the form of the key values pairs and blindly copying them into place. Of course, this condition refers to the current implementation of Clojure literal maps as array maps, which could conceivably change at any moment. However, there is an important lesson to learn from this regardless: *sometimes your best choice for a map is not a map at all*.

You've probably heard many times (including our book) that when searching for elements, arrays and lists are O(n), or proportional to the number of total elements, and maps are O(1), or constant. Big-O notation refers to a relative measure of significant algorithmic complexity and ignores tangential costs such as hash calculation, index manipulation, and the like. However, in practice it turns out that these tangential costs have an impact when dealing with sizes below a small threshold. That is, for structures sized below a certain count, the cost associated with map lookup bridges the gap between a linear search through an equally sized array or list. That's not to say that the map will be slower, instead, it allows the map and linear implementations to be comparable. The advantage of maintaining an insertion order by using a linear structure like array map can at times outweigh the slight slowdown. Therefore, it's sometimes useful to design your programs to take into account for the possibility of pointed tradeoff such as using lists or vectors in place of maps for sizes smaller than a pre-determined threshold. You can work to ensure robustness in the face of these scenarios by not presupposing a map vs. linear structure at all, but instead *working against the sequence abstraction* whenever possible.

Where have we heard that before?

```
(defn raw [n] (map keyword (map str (map char (range 97 (+ 97 n))))))
(defn mk-lin [n] (interleave (raw n) (range n)))
(defn mk-map [n] (apply hash-map (mk-lin n)))

(defn lhas [k s] (some #{k} s))
(defn mhas [k s] (s k))

(defn churn [lookup maker n]
  (let [ks (raw n)
        elems (maker n)]
   (dotimes [i 100000]
     (doseq [k ks] (lookup k elems)))))


(time (churn lhas mk-lin 5))
; "Elapsed time: 998.997 msecs"


(time (churn mhas mk-map 5))
; "Elapsed time: 133.133 msecs"
```

As we can see above, for very small sizes, the linear version is about 7.5 times slower than the version using maps which is not terribly bad. Not only does the linear map implementation provide comparable speeds at small sizes, it has the added advantage of maintaining key insertion ordering; and that truly is the sweet spot of Clojure's array maps. However, when we increase the number of elements for churn to process, the picture suddenly spoils:

```
(time (churn lhas mk-lin 10))
; "Elapsed time: 3282.077 msecs"


(time (churn mhas mk-map 10))
; "Elapsed time: 172.579 msecs"
```

Doubling the size of the structures now widens the divide in execution times between the linear and map implementations. Further increasing the sizes will eventually lead to pauses worthy of taking in a movie. It might be worth trying it yourself if for no other reason than it nicely shows that the size of your data structures will highlight the differences in their orders of magnitude.

Thinking in Maps -- A Fluent Builder for Chess Moves

People have been known to say that Java is an exceptionally verbose programming language. While this may be true when compared to the Lisp family of languages, there has been considerable mindshare devoted to devising ways to mitigate Java's verbosity. One popular technique is known as the fluent builder[16] and can be summed up as the chaining of Java methods to form a more readable, and agile, instance construction technique. In this section we will show a simple example of a fluent builder supporting the construction of chess move descriptions. We will then explain how such a technique is unneeded within Clojure and instead present an alternative approach that is simpler, concise, and more extensible. We will leverage Clojure's maps in the final solution illustrating that Java's class-based paradigm is counter to Clojure's basic principles; and often overkill for Java programs.

Imagine that we wish to represent a chess move in Java. Typically, the first approach would be to identify all of the component parts of our Move class including: from and to squares, a flag indicating if the move is a castling move, and perhaps also the desired promotion piece if applicable. Of course, there are other possible elements that might be applicable, but in order to constrain the problem we will limit our idea of a Move to those elements listed above. Of course, the next step would be to create a simple class with its properties and a set of constructors, each taking some combination of the expected properties. We would then generate a set of accessors for the properties, but not their corresponding mutators as it's probably best for the move instances to be immutable. Having created this simple class and rolled it out to the customers of our chess move API we begin to notice that our users are sending into the constructor the to string before the from string, which is sometimes placed after the promotion string, and so on. However, thanks to a design pattern called a fluent builder we can instead make the chess move API simpler to read, unambiguous, and operation order independent. After some months of intense design and weeks of development and testing we release the following elided chess move class:

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

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

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

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

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

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

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

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

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

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

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

```
        .to("e4").build();

    System.out.println(move);

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

    System.out.println(move);

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

    System.out.println(move);
}

/* prints the following:
    Move e2 to e4
    Move a1 to c1 castle
    Move a7 to a8 promote to Q
*/
```

Our constructor ambiguities have disappeared with the only trade-off being a slight increase in complexity of the implementation and the breaking of the common Java getter/setter idioms -- both of which we are wiling to deal with. However, if we had started our chess move API as a Clojure project, the code would likely be a very different experience for the end-user.

A Clojure Chess Move

To start building our chess move structure for Clojure we may make a common assumption that we need a Move class for representation. However, as we have been talking about throughout this chapter, Clojure provides a core set of composite data types and as you can guess from the title of this section, its map type is a perfect candidate for our move representation.

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

Simple no?

Coming from a background in object-oriented methodologies, we have a tendency to view problems through the lens of cumbersome class hierarchies. However, Clojure prefers simplification, providing a set of composite types perfect for representing most categories of problems typically handled by class systems. If you take a step back and think about the class hierarchies that you've built or dealt with in the past, how many of the constituent classes could have been replaced with a simple map? If your past projects are anything like ours, it's likely that the number is significant. In a language like Java it is common to represent every entity as a class, and in fact to do otherwise is either not efficiently supported, non-idiomatic, or outright taboo. However, to be a good Clojure citizen it is advisable to attempt to leverage its composite types for one very simple reason: existing functions, built on a sequence abstraction, *just work*.

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

(build-move
```

```
  :from "e7"
  :to "e8"
  :promotion \Q)

;=>gt; {:from "e2", :to "e4", :promotion \Q}
```

In two lines, we've effectively replaced the Java implementation above with an analogous, yet more flexible representation. The term domain specific language (DSL) is often thrown around to describe code like build-move, but to Clojure (and Lisps in general) the line between DSL and API is blurred. The function above makes a lot of assumptions about the form of the move pieces supplied, but it's implications cannot be denied. In our original FluentMove class we required a cornucopia of code in order to ensure the API was agnostic of the ordering of move elements; using a map we get that for free. Additionally, FluentMove, while relatively concise, was still bound by fundamental Java syntactical and semantic constraints. The Clojure build-move function on the other hand was artificially constrained as requiring alternating key-value pairs for brevity's sake. If we had instead written build-move as a macro, then there is virtually no limitation to the form that our move description might have taken; leaning closer toward the classic notion of a DSL. Finally, as author Rich Hickey himself proclaimed, any new class in general is itself an island; unusable by *any* existing code written by anyone, anywhere. So our point is this -- consider throwing the baby out with the bath water.

Separation of Concerns

Before we end this section we'd like to address an issue with both implementations that until now we've been ignoring: verification. Both FluentMove and build-move make enormous assumptions about the form of the data supplied to them and do no validation of the input. For FluentMove object-oriented principles dictate that the validation of a well-formed move (not a legal move mind you) should be determined by the class itself. There are a number of problems with this approach the most obvious being that in order for FluentMove to determine if it is a well-formed move it needs some information about the rules of chess; for example, a move cannot be both a castle *and* a piece promotion. We can certainly rewrite FluentMove to throw an exception to prevent such a case from being entered, but the root problem still remains -- FluentMove instances are too smart. Conversely, you could take the opposite extreme and require the client to check the validity of the moves, but if they could do that in the first place, then they likely wouldn't need your code at all. Perhaps you don't see this as a problem, but if we were to extend our API to include other aspects of the game of chess, then we'll find that using this approach requires that little bits of overlapping chess rules need to be scattered throughout your class hierarchy. Really what we wanted from the start is for our move representation to be a value, devoid of associated validation behavior.

By viewing the move structure as a simple value map, our Clojure code provides some freedom in the implementation of a total solution. That is, we decouple the value and its validation behavior allowing for the latter to be composed from numerous function, each swapped in and out as necessary.

We have covered the basics of Clojure maps in this section including common usage and construction techniques. Clojure maps, minus a couple implementation details, should not be surprising to anyone. It will take some time to grow accustomed to dealing with immutable maps, but in time even this nuance will become second nature.

Clojure favors simplicity in the face of growing software complexity. If our problems are easy solved by sequential and collection abstractions then it is those exact abstractions that should be used. Much of the problems of software can indeed be modeled on such simple abstractions, yet we continue to build monolithic class hierarchies in attempts to deceive ourselves into believing that we're mirroring

the "real world" -- whatever that means. Perhaps it's time to realize that we no longer need to layer our self-imposed complexities on top of software solutions that are already inherently complex. Not only does Clojure provide the sequential, set, and map types useful for pulling ourselves from the doldrums of software complexity, but it is also optimized for dealing with them.

---

[1] See "Java Concurrency in Practice" by Brian Goetz for more details.

[2] We're intentionally glossing over Clojure's features that support mutability such as reference types and transients in order to keep this section focused.

[3] Reginald Braithwaite discusses this language-level separation of concerns in his excellent blog post "Why Why Functional Programming Matters Matters" at http://tinyurl.com/yyfpmm.

[4] For more information about these statements, see Henry Baker's essay "Equal Rights for Functional Objects or, The More Things Change, The More They Are the Same".

[5] An excellent example being http://blog.higher-order.net/2009/02/01/understanding-clojures-persistentvector-implementation/

[6] Clojure hash-maps, hash-sets, and vectors all have up to 32 branches per node. This results in dramatically shallower trees, and therefore faster lookups and updates.

[7] Clojure's sorted-map and sorted-set do use a binary tree internally, but implement red-black trees to keep the left and right sides nicely balanced.

[8] Vectors can't be concatenated any more efficiently than O(n), but later we will cover a data structure that can. Finger trees, a possible future addition to Clojure, can be looked up by numerical index a bit like vectors, and yet also be concatenated in logarithmic time.

[9] Several operations on Clojure's persistent data structures are described in our book as "essentially constant time". In all cases these are O(log32 n).

[10] Nested vectors are far from the most efficient way to store or process vectors, but they are convenient to manipulate in Clojure and so make a good example here. More efficient options include a single vector, arrays, or a library for matrix processing like Colt or Incanter at http://incanter.org.

[11] The conj function also work all of Clojure's other persistent collection types, even if they don't implement clojure.lang.IPersistentStack.

[12] ...the most natural *tail-recursive* algorithm anyway.

[13] Clojure 1.0 also had a LazilyPersistentVector for performance reasons, but this is no longer necessary because PersistentVector now supports transient.

[14] Although with the pervasiveness of the map literal, the ubiquity may instead fall to the array map.

[15] The default sorted-map comparison function is a static member of Clojure's RT class. You can try it out directly like this: (.compare clojure.lang.RT/DEFAULT_COMPARATOR "abc" "bac") Clearly, based on the ugliness of that call, this isn't something you're expected to do very often.

[16] Based on the original concept "Fluent Interface", coined by Martin Fowler. http://martinfowler.com/bliki/FluentInterface.html

- **This article is part of a featured topic series on** Java
- See more Java content at: http://www.infoq.com/java

- Other recent content items in this topic
  - Book Review and Interview: Java Performance, by Charlie Hunt and Binu John
  - Martin Odersky on Typesafe Stack and the Future of Scala

5 comments

Watch Thread Reply

Example code missing or messed up in "Designing a Persistent Toy" example? by Mike Bria Posted 08/05/2010 05:19
Code examples are mangled by Brian Gruber Posted 09/05/2010 08:32
Object by Greg Helton Posted 09/05/2010 02:06
Great Article by Richard Claxton Posted 16/06/2010 08:19
gt; by Roger Pack Posted 16/06/2011 11:51
Sort by date descending

1. Back to top

Example code missing or messed up in "Designing a Persistent Toy" example?

08/05/2010 05:19 by **Mike Bria**

The example blocks highlighted the "xconj" function all appear the same. Am I missing something, or is the example code wrong?

Reply

2. Back to top

Code examples are mangled

09/05/2010 08:32 by **Brian Gruber**

They look like they were eaten by an overzealous html-ifier.

Reply

3. Back to top

Object

09/05/2010 02:06 by **Greg Helton**

The article uses 'object' quite a lot even thought the language isn't OO. I can't tell if the authors are using the term 'object' to refer to something in Clojure that is never defined or if they are referring to the backing Java components or if all the object oriented talk is just to explain immutability in OO terminology as a metaphor for what immutability in Clojure is similar to?

Also, the authors talk quite a bit about immutability AS IT DOES NOT pertain to Clojure which leads them to write this sentence: "However, in order to foster an air of immutability in our own systems it is of the utmost importance to crate a facade of immutability". I'm not sure that knowing this helps me with Clojure.

The authors think OO developers "have a tendency to view problems through the lens of cumbersome class hierarchies" which I don't believe. It is a nice, sweeping indictment but even the bad code I see doesn't have cumbersome class hierarchies.

All in all, the writing is sometimes imprecise and bloviated and often meanders.

Reply

4. Back to top

Great Article

16/06/2010 08:19 by **Richard Claxton**

This is a really good Article, the last paragraph is especially relevant.
I wrote a telecommunications system for the Swedish police in a OO language, after it was finished I remarked to another developer in my team that all we had developed was a concurrent list handing system.

Reply

5. Back to top

gt;

16/06/2011 11:51 by **Roger Pack**

Why are there gt; in the code snippets?

Reply