Using Collections in Clojure

Zachary Bennett
Software Engineer

@z_bennett_ zachbennettcodes.com

Built-in Collections

Lists

Vectors

Sets

Maps

Collections are immutable!

Collection Equality

Lists/Vectors

Contain the same elements beginning to end

Sets

Contain the exact same set of elements as one another

Maps

Contain the same set of key/value pairs as one another

Sequence abstraction!

Core Sequence/Collection Functions

Function Signature	Purpose
seq first(seq)	Returns the first element within a sequence.
seq rest(seq)	Returns the rest of the sequence after the first element.
seq map(fn, seq)	Applies the given function “fn” to each element in the given sequence.
value reduce(fn, seq)	Applies the given reducer function to the given sequence.
seq filter(fn, seq)	Filters a given sequence using the boolean predicate function supplied.

Core Sequence/Collection Functions

Function Signature	Purpose
seq sort(seq) seq sort(comparator, seq)	Sorts the given sequence by the “compare” function or the given comparator function if present
seq sort-by(keyfn, seq) seq sort-by(keyfn, comparator, seq)	Sorts the given sequence via the value produced by applying the given “keyfn” to the sequence in combination with an optional comparator function.
seq take(n, seq)	Produces a new sequence containing the first “n” elements from the given sequence
seq into(to, from)	Puts all of the items in the “from” sequence into a new sequence defined by the “to” argument
seq conj(seq, x)	Adds the value of “x” to the given sequence. Where “x” is added to the sequence is dependent upon the data type of the given sequence

Lists

Lists are sequential linked lists!

```
;; Example of linked list functionality


(def my-list `(1 2 3 4))


(conj my-list 5)


;; results in => (5 1 2 3 4)
```

- Define a list
- Remember, “conj” works on all of the major sequences
- For the list data type, “conj” adds elements to the head of the list

```
;; You can also use a list as a stack

(def my-list `(1 2 3 4))

(peek my-list)

;; results in => 1

(pop my-list)

;; results in => (2 3 4)
```

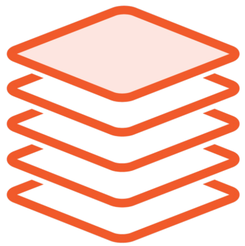
- Peek lets us access this list as a stack
- It returns the first element in the list
- Pop lets us take the first item off of the list

Core List Functions

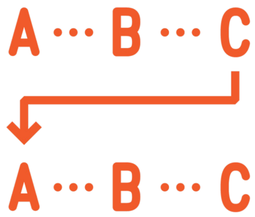
Function Signature	Purpose
list list(& items)	Creates a list from the given arguments
boolean list?(seq)	Returns true if the given argument is a list, i.e. it implements the core IPersistentList interface

When To Use Lists

Your data is ordered sequentially in some way



You don’t need indexed/fast access



You find yourself adding and fetching items to and from the front or the back of your sequence frequently

Vectors

Vectors are indexed!

```
;; Example of vector array access


(def my-vec [1 2 3 4])


(get my-vec 0)


;; results in => 1
```

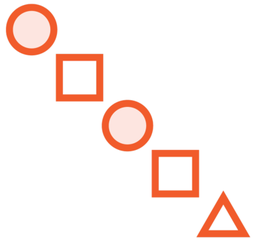
- Define a vector

- The “get” function is used to access elements in a vector via a numerical index

Core Vector Functions

Function Signature	Purpose
vector vector(& items)	Creates a vector from the given arguments
boolean vector?(seq)	Returns true if the given argument is a vector, i.e. it implements the core IPersistentVector interface
value get(vector, index)	Returns the value at the given index or nil if there is no value present
vector assoc(vector, index, value)	Puts the given value inside the given vector at the given index
vector subvec(vector, start, end) vector subvec(vector, start)	Creates a subset of a vector given start and end indexes. If no end index is supplied, the last index of the vector will be used.

When To Use Vectors



Your data is ordered sequentially in some way

You need indexed/fast access

You find yourself needing to access elements within your ordered collection in an arbitrary or random order

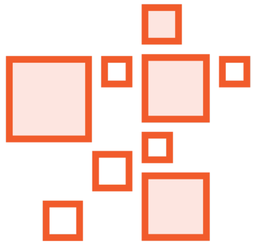
Sets

Collection of unordered, unique elements

Core Set Functions

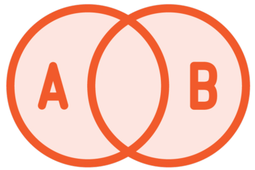
Function Signature	Purpose
set conj(set, element)	Adds an element to a set
boolean contains?(seq, element)	Returns true if the given set contains the given element
set disj(set, element)	Removes the given element from the given set
set union(set1, set2)	Combines two sets – producing the union of the two
set difference(set1, set2)	Creates a set that consists of all of the elements of set1 that are not present within set2
set intersection(set1, set2)	Creates a set that consists of elements that are present within both sets
boolean subset?(set1, set2)	Returns true if all elements in set1 are present within set2

When To Use Sets



Your data is unordered

You are very concerned with the uniqueness of values in your collection



You want to take advantage of the mathematical concepts

Maps

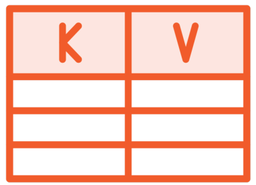
Collection of unordered, key/value pairs

Core Map Functions

Function Signature	Purpose
map assoc(map, key, value)	Adds a key/value pair to the given map – producing a new map
boolean contains?(map, key)	Returns true if the given map contains the given key
vector find(map, key)	Returns the key/value pair within a vector if there is a matching key
list keys(map)	Returns a list of keys within the given map
list vals(map)	Returns a list of values within the given map
map dissoc(map, key)	Returns a new map minus the key/value pair found via the “key” argument
map merge(map1, map2)	Returns a new map with the result of merging map2 with map1

When To Use Maps

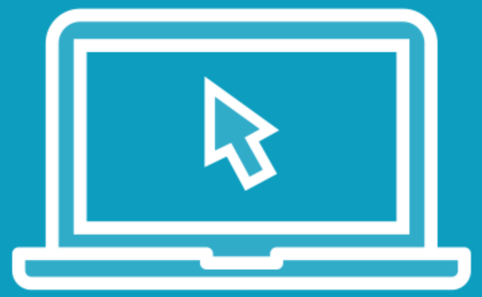
Your data is unordered



Modeling domain problems/creating custom data types

You want fast search, insertion and deletion of elements

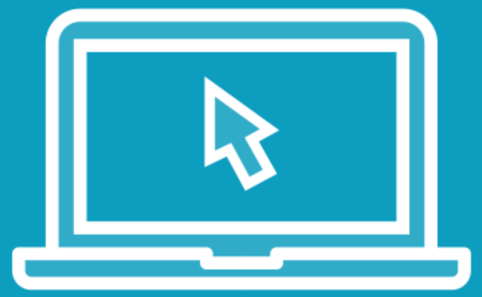
Demo



Sequential Collections

- Lists
- Vectors

Demo



Hashed Collections

- Sets
- Maps