

JavaScript Immutability

Don't Go Changing

Mark Volkmann, Object Computing, Inc.

Email: mark@ociweb.com

Twitter: @mark_volkman

GitHub: mvolkmann

Website: <http://ociweb.com/mark>

<https://github.com/mvolkmann/react-examples/Immutable>



OCI | WE ARE
SOFTWARE
ENGINEERS.

Intro.

- What is **OCI**?
 - new home of **Grails**,
“An Open Source high-productivity framework for building fast and scalable web applications”
 - Open Source Transformation Services, IIoT, DevOps
 - offsite development, consulting, training
 - **handouts** available (includes Grails sticker)
- What does this talk have to do with **Billy Joel** and the song “Just the Way You Are”?
- **Three parts**
 - What is immutability and how is it implemented?
 - What are the options in JavaScript?
 - Overview of API for one option and examples

Immutability Defined

- Immutable values cannot be modified after creation
- In many programming languages, strings are immutable
 - methods on them return new versions rather than modifying original
- Data structures can also be immutable
- Rather than modifying them, create a new version
- Naive approach - copying original and modify copy
- We can do better!

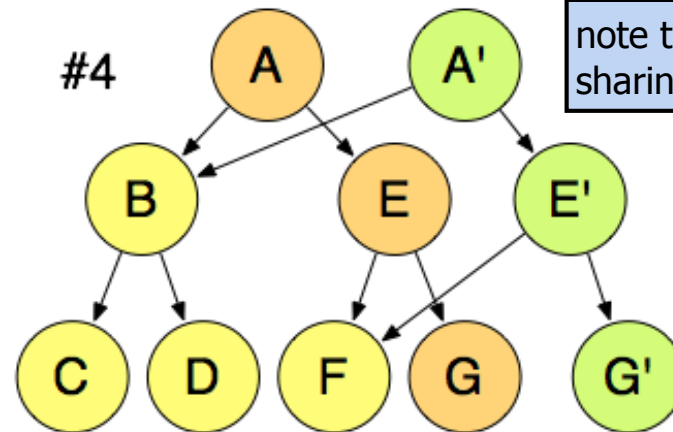
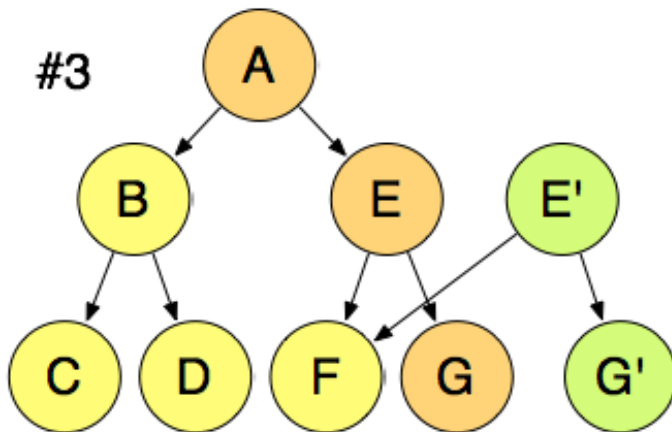
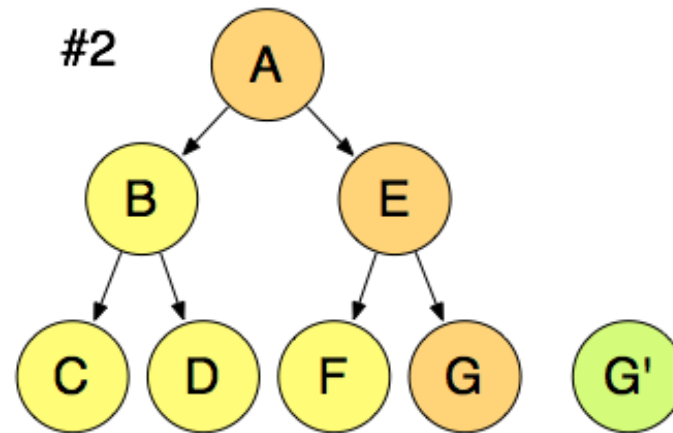
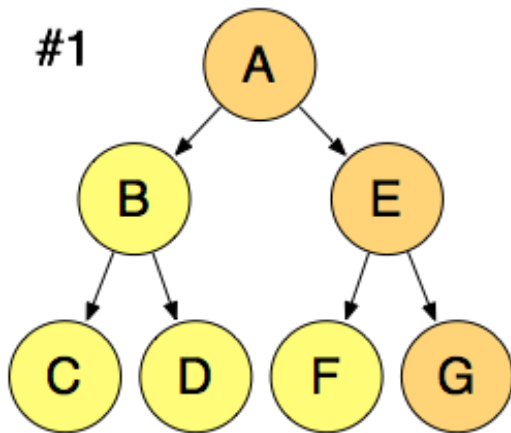
Persistent Data Structures

It's not necessary to understand how these work to take advantage a library that uses them.

- Wikipedia says “**a data structure that always preserves the previous version of itself when it is modified**”
- Uses **structural sharing** to efficiently create new versions of data structures like lists and maps
- Typically implemented with
 - **index tries**
 - **hash array map tries** (HAMT)
- **Slower** and uses **more memory** than operating on mutable data structures
 - but fast enough for most uses
- Explained well in video “Tech Talk: Lee Byron on Immutable.js”
 - Lee Byron is at Facebook
 - <https://www.youtube.com/watch?v=kbnUIhsX2ds&list=WL&index=34>
- Uses **Directed Acyclic Graphs** (DAGs)

DAGs

- Can be used to represent a list
- Diagrams show new version of list created for new value of node G



note the structural sharing that results

every time a node is added or modified, make a copy of all ancestor nodes and return the top one

Tries

- A **trie** is a special kind of DAG
 - name taken from “re**TRIE**val”
 - correct pronunciation is “tree”, but many say “try” because computer science already has something called a tree
- We’ll discuss two types
 - **index trie** used to model arrays
 - **hash array mapped trie** (HAMT) used to model sets and maps

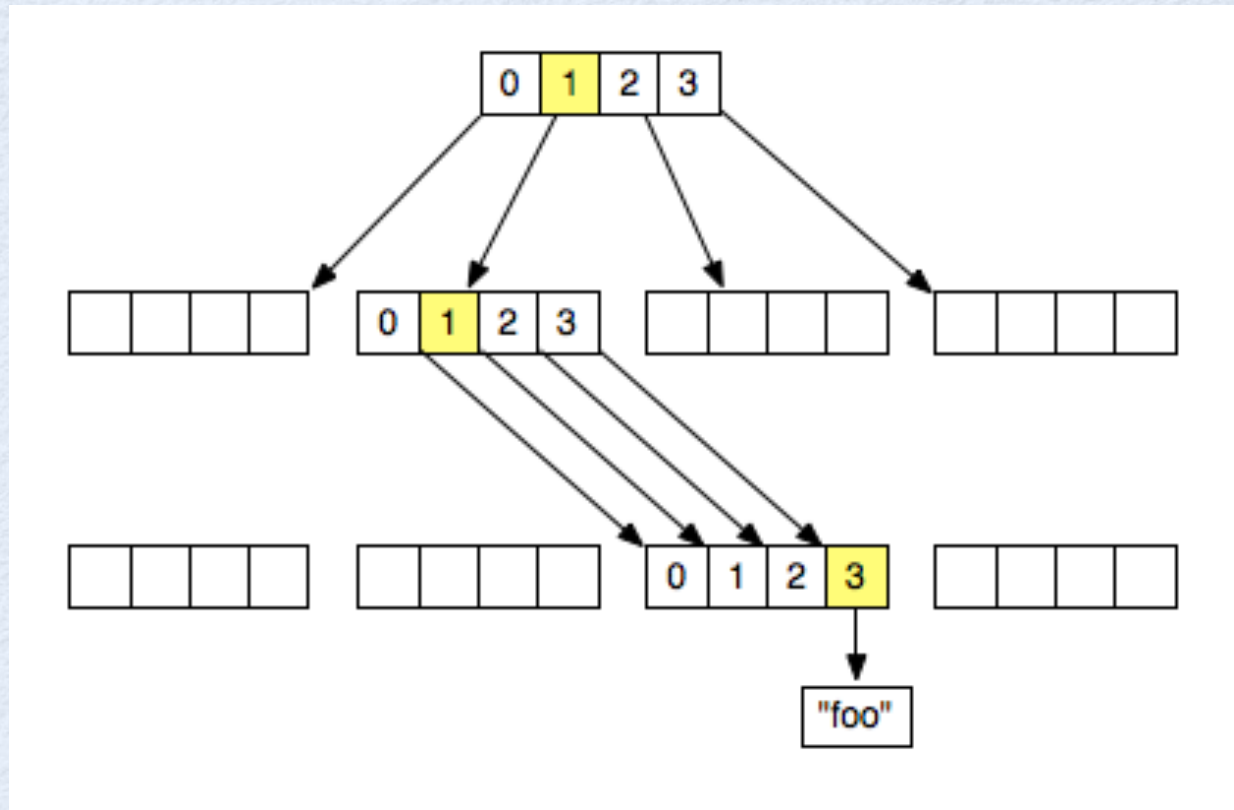
Index Trie ...

- Nodes are fixed-size arrays of pointers to other nodes or values

- store value "foo" at index 53
- 53 in binary is 110101
- starting from least significant bits, the pairs are 01, 01, and 11 or node indexes 1, 1, and 3

least significant bits tend to be more random

- use same process to lookup a value at a given index
- typically node size is 64 instead of 4 to match hardware "word" size



... Index Trie

- To set a new value at a given index, use the DAG approach described earlier to create new versions of existing nodes so those remain unchanged
- Ditto for marking a value “undefined” or popping a value from end
- Values can only be efficiently removed or inserted at end
 - not at beginning or in middle because indexes of other values would have to change

Recall that JS arrays are modeled as objects. The `Array shift` method is not efficient. See pseudocode at <https://tc39.github.io/ecma262/#sec-array.prototype.shift>.

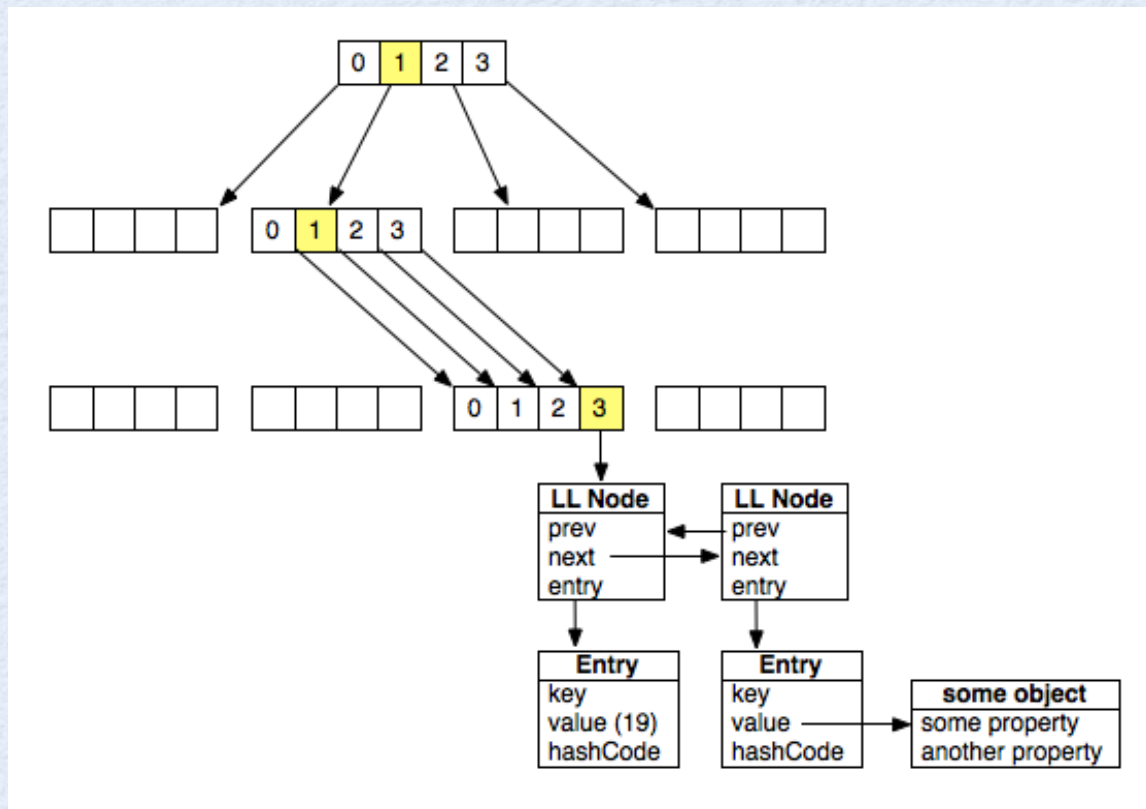
Hash Array Mapped Trie ...

- Used to model sets, maps, and objects
 - maps are collections of key/value pairs
 - think of sets as collections of keys
- Similar to an index trie
- Invented by Phil Bagel and iterated on by Rich Hickey
- Instead of array indexes, hash codes of keys are used
 - need way to compute hash code for strings
 - no hash code functions are provided by JavaScript
 - some approaches are documented at <http://erlycoder.com/49/javascript-hash-functions-to-convert-string-into-integer-hash->
 - if keys of other primitive types (boolean and number) are allowed, can use `toString` method to convert to a string and hash that
 - if object keys are allowed, hash code can be computed by some combination of hash codes of its property values

... Hash Array Mapped Trie ...

- Node array values (“slots”) have three possibilities
 - **empty** (undefined)
 - reference to another **trie node**
 - reference to **linked list node** see picture on next slide
 - holds **previous/next** references to other linked list nodes and **entry** reference
 - previous/next references support having a linked list of objects for when more than one key has **same hash code** (shouldn’t happen frequently, but can’t rule out)
 - **entry objects** hold **key, value,** and **hash code** of key
 - when traversal leads to a list of objects, linear search finds correct one by key
- Adding or removing an entry
 - results in a new HAMT that uses structural sharing with previous version
 - when copying a trie node, can copy references to existing linked list nodes

... Hash Array Mapped Trie



- Level optimization
 - when storing a value, if an empty slot is reached, store value there, using a subset of hash code bits
 - later if another value ends up at that same slot, move both along deeper until subset of hash code bits differ
 - but need to compare key value on lookup
- Existing value optimization
 - if a key is set to its existing value, return same structure

For even more detail, see the book
"Purely Functional Data Structures"
by Chris Okazaki

Immutable Pros

- **Some side effects avoided**

- can pass immutable values to a function and know it cannot modify them

- **Pure functions easier to write**

- can pass an object and return an efficiently modified version

- **Fast change detection**

- rather than deep comparison, can just compare object references
- in JavaScript, use ===

- **Immutable data can be safely cached**

- no possibility of code changing it after it has been cached

- **Easier to implement undo**

- keep a list of past values and reset to one of them
- but doesn't undo changes to persistent stores like databases

- **Concurrency**

- can share data between threads without concern over concurrent access not a concern in JavaScript

Immutable Cons

- **Performance**

- takes longer to create a new version of a persistent data structure than to mutate a mutable data structure like an array or map
- takes longer to lookup a value in a persistent data structure than in a mutable data structure like an array or map

- **Memory**

- structural sharing uses more memory than mutable data structures

- **Learning curve**

- can't use standard JavaScript API for collections (**Array**, **Object**, **Set**, **Map**)
- must learn new API

React and Immutability



- React (“JavaScript library for building user interfaces”) favors immutable objects
- Should not modify properties in state objects
- Instead, create a new object and pass to **setState** method of components
 - or use Redux to manage state
- Manually creating a modified copy of state is tedious, error prone, and expensive in terms of memory
- Better to use an immutability library that utilizes structural sharing

Options ...

- **Be careful**
 - write code that avoids mutations
- **Immutability helpers**
 - from React team
 - <https://facebook.github.io/react/docs/update.html>
 - **doesn't use structural sharing** (a.k.a. persistent data structures)
- **seamless-immutable**
 - from Richard Feldman
 - <https://github.com/rtfeldman/seamless-immutable>
 - **doesn't use structural sharing**

... Options

- **Mori**

- from David Nolan
- <https://github.com/swannodette/mori>
and <http://swannodette.github.io/mori/>
- **uses structural sharing**
- Clojure persistent data structures ported to JavaScript

- **Immutable**

- from Lee Byron at Facebook
- <https://facebook.github.io/immutable-js/>
- **uses structural sharing**
- great overview from React.js Conf 2015
“Immutable Data and React” by Lee Byron of Facebook
<https://www.youtube.com/watch?v=I7IdS-PbEgI>
- **we will mainly focus on this**

Being Careful

This road leads to madness!

- Add element to end of array
 - ES6: `newArr = [...oldArr, elem]`
- Insert element at index in array
 - ES6: `newArr = [...oldArr.slice(0, index), elem, ...oldArr.slice(index)]`
- Remove element at index from array
 - ES6: `newArr = [...oldArr.slice(0, index), ...oldArr.slice(index + 1)]`
- Modify element at index in array
 - ES6: `newArr = [...oldArr.slice(0, index), newElem, ...oldArr.slice(index + 1)]`
- Modify or add property in object
 - ES6: `newObj = Object.assign({}, oldObj, {propName: propValue});`
 - ES7: `newObj = {...oldObj, propName: propValue};` uses object spread operator

consider using **deep-freeze** to prevent accidental mutations
<https://github.com/substack/deep-freeze>

Immutability Helpers

- <https://facebook.github.io/react/docs/update.html>
 - see examples here
- Install with `npm install --save-dev react-addons-update`
- Usage

```
const update = require('react-addons-update');
const newObj = update(oldObj, changes);
```
- Object commands
 - `$set: value` - replaces target value with specified `value` (no `$unset`, but it has been proposed)
 - `$merge: obj` - replace target object with result of merging properties in `obj` with current value
 - `$apply: fn` - replaces target value with result of `fn` when passed current value
- Array Commands
 - `$push: arr` - adds all elements in `arr` to end of target array
 - `$unshift: arr` - adds all elements in `arr` to beginning of target array
 - `$splice: arr` - each `arr` element is an array of splice arguments; creates new array from target by calling splice with each set of arguments

seamless-immutable

- **Creates objects that are backward-compatible with JS Arrays and Objects**
- Efficiently copies objects by reusing existing nested objects whose properties aren't changed
- Operates differently depending whether built for development or production
 - **development** - objects are frozen; overrides methods that normally mutate to throw
 - **production** - assumes code has been tested in development mode and favors performance by not doing these things
- **Immutable** function takes any object and returns a backward-compatible, immutable version
- **Doesn't work with objects that contain circular references**
- Adds methods to immutable objects: **merge, without, asMutable**
- Adds methods to immutable arrays: **flatMap, asObject, asMutable**

Mori

- **Uses Clojure terminology**
 - such as `assoc`, `dissoc`, `conj`, `transduce`, and `vector`
- Used in ClojureScript
 - can also be used in JavaScript
- **Uses structural sharing**
- Faster than other libraries
- Has a functional API
 - data structures are passed to functions rather than having methods on them in OO-style
- Larger library than Immutable
 - after gzipping both, Mori **2.4 times as large as Immutable**

Immutable

- “Inspired by inspired by Clojure, Scala, Haskell and other functional programming environments”
- API mirrors ES6 **Array**, **Map**, and **Set** methods
 - but methods that mutate in ES6 return an immutable copy instead
 - ex. Array methods **push**, **pop**, **unshift**, **shift**, **splice**
- **Uses structural sharing**
 - makes copying more efficient in both performance and memory usage
- Provides many immutable classes
 - listed on “Collection Types” slide ahead
- Remaining slides focus on this library

Setup

- To install,
`npm install --save immutable`
- To use in ES5 browser code,
`<script src="node-modules/immutable/dist/immutable.min.js"></script>`
- To use in ES6 browser code,
`import Immutable from 'immutable';`
- To use in Node code,
`const Immutable = require('immutable');`

Collection Types

see documentation at <http://facebook.github.io/immutable-js/docs>

- **Map** and **OrderedMap**
 - similar to ES6 **Map**
 - **OrderedMap** iteration order matches order added
- **List**
 - similar to JavaScript **Array**
- **Set** and **OrderedSet**
 - similar to ES6 **Set**
 - **OrderedSet** iteration order matches order added
- **Stack**
 - singly linked list
 - efficient addition and removal at front
- **Record**
 - “creates a new class which produces **Record** instances ... similar to a JS object”
- **Iterables**
 - **Iterable**, **KeyedIterable**, **IndexedIterable**, **SetIterable**
 - all are ES6 iterables
- **Sequences**
 - **Seq**, **KeyedSeq**, **IndexedSeq**, **SetSeq**
 - support lazy evaluation
- **Collection base classes**
 - **Collection**, **KeyedCollection**, **IndexedCollection**, **SetCollection**

Nesting

- Can nest immutable objects
 - ex. immutable `Map` with properties whose values are immutable `List` objects
- It can be confusing and error prone to use non-immutable values (such as standard JavaScript objects and arrays) as values in immutable structures
 - be consistent!

JS to Immutable

- To convert an `Object` or `Array` to an immutable `Map` or `List`,
`const immObj = Immutable.fromJS(mutObj);`
- To customize the conversion and choose the collection types to be used,
`const immObj = Immutable.fromJS(mutObj, (key, value) => {
 // Only called for non-primitive values.
 // value will be a Seq object.
 // Return an immutable object.
});`

Immutable to JS

- To convert an immutable object to a JavaScript **Object** or **Array**,
`const mutObj = immObj.toJS();`
- Resist the urge to do this just so values can be accessed in a standard JavaScript way
 - less efficient than using methods on immutable objects

Working With Maps ...

- To create

- `const map = Immutable.Map();`
 - can pass many kinds of things to initialize
- `const map = Immutable.fromJS(jsObject);`
 - makes deep copy where all values are immutable
 - objects -> `Maps`; arrays -> `Lists`

- To set top-level key value

- `const newMap = map.set(key, value);`

- To set deeper key value

- `const newMap = map.setIn([key-path], value);`

`key-path` is an ordered array of keys;
ex. `['work', 'address', 'city']`

- To get top-level key value

- `const value = map.get(key);`

- To get deeper key value

- `const value = map.getIn([key-path]);`

... Working With Maps ...

- To update top-level key value
 - `const newMap = map.update(key, fn);`
 - value at `key` is passed to `fn` and return value becomes new value
- To update deeper key value
 - `const newMap = map.updateIn([key-path], fn);`
 - value at `key-path` is passed to `fn` and return value becomes new value
- To delete top-level key/value pair
 - `const newMap = map.delete(key);`
- To delete deeper key/value pair
 - `const newMap = map.deleteIn([key-path]);`

... Working With Maps

- To iterate over

- keys - `const iter = map.keys();`

- values - `const iter = map.values();`

- entries - `const iter = map.entries();` entries are [key, value] arrays

- value returned from each of these is an ES6 iterable, so can use with ES6 for-of loop

```
for (const entry of teams.entries()) { ... }
```

- There are MANY more methods on `Map` listed later

- Working with other kinds of collections is similar

Map API Examples

```
import Immutable from 'immutable';
```

```
let person = Immutable.fromJS({  
  name: 'Moe Howard',  
  address: {  
    street: '123 Some Street',  
    city: 'Somewhere',  
    state: 'MO',  
    zip: 12345  
  }  
});
```

```
person = person.set('name', 'Larry Fine');  
person = person.setIn(['address', 'city'], 'Los Angeles');  
console.log('name =', person.get('name'));  
console.log('city =', person.getIn(['address', 'city']));  
person = person.deleteIn(['address', 'street']);  
person = person.updateIn(['address', 'zip'], zip => zip + 1);  
console.log(person.toJS());
```

Output

```
name = Larry Fine  
city = Los Angeles  
{  
  name: 'Larry Fine',  
  address: {  
    zip: 12346,  
    city: 'Los Angeles',  
    state: 'MO'  
  }  
}
```

can chain all calls that
create a new version

Multiple Mutations

- When modifying multiple properties in an immutable object, it can be more efficient to make them on a mutable version and then create an immutable version from that
 - avoids creating multiple new, immutable objects
- **withMutations** method does this
 - call on an immutable object
 - pass a function that will be invoked with a mutable version of it
 - returns a new, immutable object

```
person = person.withMutations(mutPerson =>
  mutPerson.set('name', 'Larry Fine').
    setIn(['address', 'city'], 'Los Angeles').
    deleteIn(['address', 'street']).
    updateIn(['address', 'zip'], zip => zip + 1));
```

alternate version of
code on previous slide

Working With Lists

- See example code on next slide
- `List` class has MANY more methods than are demonstrated

List API Examples

```
let numbers = Immutable.fromJS([10, 20, 30]);
console.log(numbers.get(1)); // 20
console.log(numbers.first()); // 10
console.log(numbers.last()); // 30
console.log(numbers.has(2), numbers.includes(2)); // true, false
console.log(numbers.has(20), numbers.includes(20)); // false, true
numbers = numbers.push(40); // [10, 20, 30, 40]
numbers = numbers.pop(); // [10, 20, 30]
numbers = numbers.unshift(0); // [0, 10, 20, 30]
numbers = numbers.shift(); // [10, 20, 30]
numbers = numbers.set(1, 7); // [10, 7, 30]
numbers = numbers.delete(1); // [10, 30]
numbers = numbers.update(1, n => n * 2); // [10, 60]
numbers = numbers.splice(1, 0, 20, 30, 40, 50); // [10, 20, 30, 40, 50, 60]

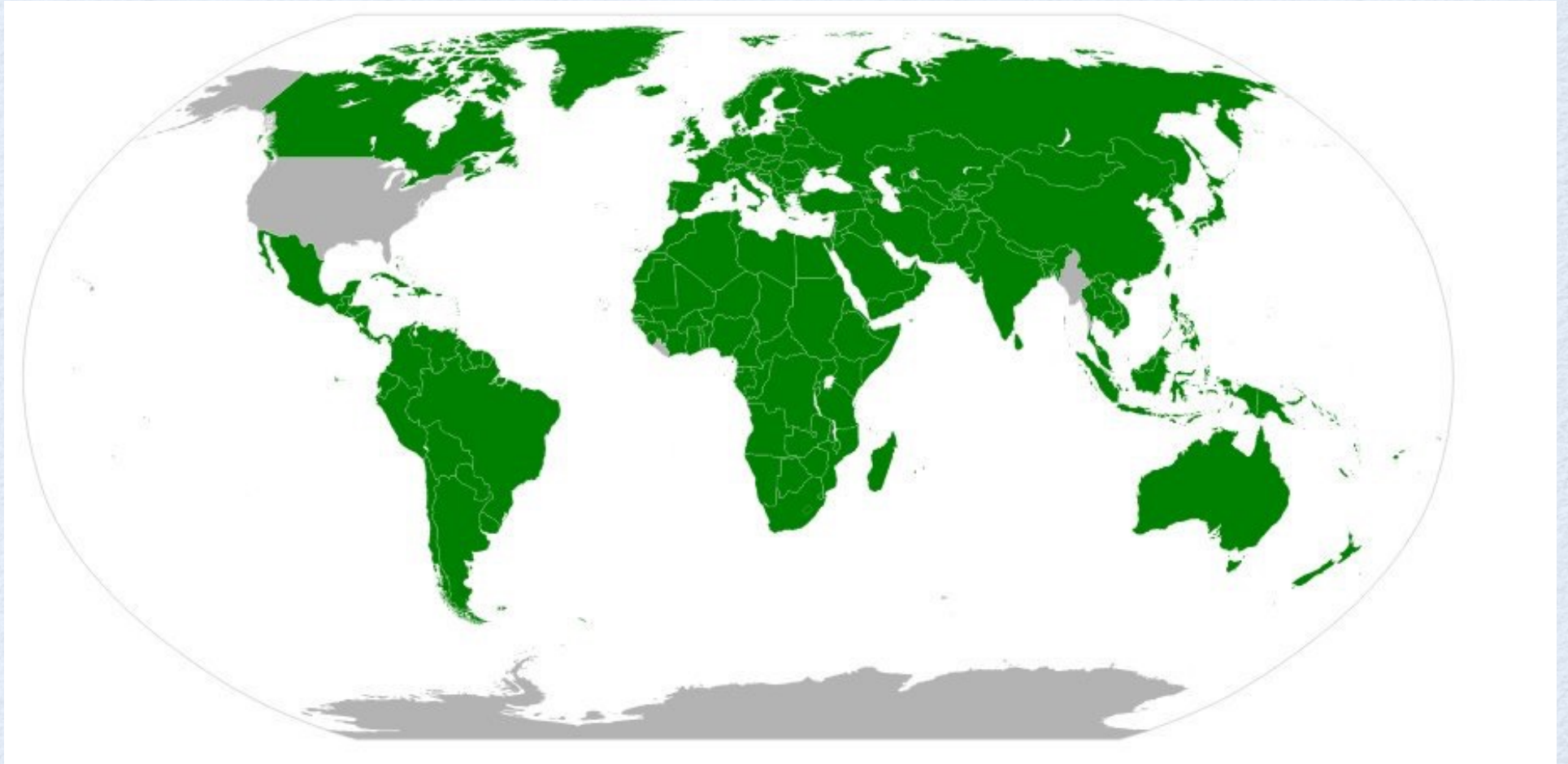
let people = Immutable.fromJS([
  {name: 'Mark', height: 74, occupation: 'software engineer'},
  {name: 'Tami', height: 64, occupation: 'vet receptionist'}
]);
console.log(people.getIn([0, 'occupation'])); // software engineer
people = people.setIn([1, 'occupation'], 'retired'); // Tami is retired
people = people.deleteIn([1, 'occupation']); // Tami has no occupation
people = people.updateIn([1, 'height'],
  height => height + 1); // Tami's height is 65

// Lists are iterable!
for (const person of people) {
  console.log(person);
}
```

`has` method looks for index;
`includes` method looks for value

74" \approx 188 cm
64" \approx 163 cm

Metric System Usage



Seqs

- Represent a sequence of values
 - backed by another data structure when created with `toSeq`, `toKeyedSeq`, `toIndexedSeq`, and `toSetSeq` methods
 - can create directly with `Seq`, `KeyedSeq`, `IndexedSeq`, and `SetSeq` constructors
 - values can be primitives and objects, including other immutable data structures
- Immutable
 - many methods create a new, immutable version:
`concat`, `map`, `reverse`, `sort`, `sortBy`, `groupBy`, `flatten`, `flatMap`
 - many methods create immutable subsets:
`filter`, `filterNot`, `slice`, `rest` (all but first), `butLast`,
`skip`, `skipLast`, `skipWhile`, `skipUntil`,
`take`, `takeLast`, `takeWhile`, `takeUtil`
- Lazy
 - “does as little work as necessary to respond to any method call”
 - see example on next slide

`Seq` has a large API.
This scratches the surface.

Seq Example

Range returns an **IndexedSeq** of numbers from start (inclusive, defaults to 0) to end (exclusive, defaults to infinity), by step (defaults to 1)

Infinity is a predefined global variable in JavaScript

```
const result =  
  Immutable.Range(1, Infinity). // all positive integers  
  filter(n => n % 7 === 0). // all numbers divisible by 7  
  take(3). // just first three: 7, 14, 21  
  map(n => n * 2). // double them: 14, 28, 42  
  reduce((sum, n) => sum + n); // sum them: 84
```

Comparing Objects

- To determine if two immutable objects contain the same data,
`Immutable.is(immObj1, immObj2)`
 - performs a **deep equality check** that works as expected when comparing nested, immutable objects
 - unlike `Object.is` added in ES6 which does **not** perform a **deep** equality check
- If one immutable object was created by potential modifications on another, this can be simplified to
`immObj1 === immObj2`

API Summary

- The remaining slides summarize the methods available in each of the collection types
- It's a large API!

skip to slide 57

Persistent Changes



	Map/OrderedMap	List	Set/OrderedSet	Stack	Seq
set	X	X			
delete	X	X	X		
clear	X	X	X	X	
update	X	X			
merge	X	X			
mergeWith	X	X			
mergeDeep	X	X			
mergeDeepWith	X	X			
push		X		X	
pop		X		X	
unshift		X		X	
shift		X		X	
setSize		X			
add			X		
union			X		
intersect			X		
subtract			X		
pushAll				X	
unshiftAll				X	

Deep Persistent Changes



	Map/OrderedMap	List	Set/OrderedSet	Stack	Seq
setIn	X	X			
deleteIn	X	X			
updateIn	X	X			
mergeIn	X	X			
mergeDeepIn	X	X			

Transient Changes



	Map/OrderedMap	List	Set/OrderedSet	Stack	Seq
withMutations	X	X	X	X	
asMutable	X	X	X	X	
asImmutable	X	X	X	X	

Conversion to Seq



	Map/OrderedMap	List	Set/OrderedSet	Stack	Seq
toSeq	X	X	X	X	
toKeyedSeq	X	X	X	X	
toIndexedSeq	X	X	X	X	
toSetSeq	X	X	X	X	
fromEntrySeq		X		X	

Value Equality



- All collection types support these methods
 - `equals`
 - `hashCode`

Reading Values



	Map/OrderedMap	List	Set/OrderedSet	Stack	Seq
get	X	X	X	X	X
has	X	X	X	X	X
includes	X	X	X	X	X
first	X	X	X	X	X
last	X	X	X	X	X
peek				X	

Reading Deep Values



- All collection types support these methods
 - `getIn`
 - `hasIn`

Conversion to JavaScript Types



- Can convert all immutable structures back to standard JS objects
- **toObject** method
 - returns JS object created from top-level properties of immutable object (shallow)
- **toArray** method
 - returns JS array created from top-level properties of immutable object (shallow)
- **toJS** method
 - like **toObject**, but deep
- **toJSON** method
 - just an alias for **toJS**

Conversion to Collections



- All collection types support these methods
 - `toMap`
 - `toOrderedMap`
 - `toSet`
 - `toOrderedSet`
 - `toList`
 - `toStack`

Iterators and Iterables



- All collection types support these methods
 - `keys`
 - `values`
 - `entries`
 - `keySeq`
 - `valueSeq`
 - `entrySeq`

Sequence Algorithms



- All collection types support these methods
 - `map`
 - `filter`
 - `filterNot`
 - `reverse`
 - `sort`
 - `sortBy`
 - `groupBy`

Side Effects



- All collection types support this method
 - **forEach**

Creating Subsets



- All these methods are available on all collection types and return an **Iterable** of same type over a subset of the elements
- **slice** (**begin**, **end**) - from begin to just before end
- **rest** () - all but first
- **butLast** () - all but last
- **skip** (**n**) - all but first **n**
- **skipLast** (**n**) - all but last **n**
- **skipWhile** (**predicate**) - all starting with first where **predicate** returns **false**
- **skipUntil** (**predicate**) - all starting with first where **predicate** returns **true**
- **take** (**n**) - first **n**
- **takeLast** (**n**) - last **n**
- **takeWhile** (**predicate**) - initial elements while **predicate** returns **true**
- **takeUntil** (**predicate**) - initial elements until **predicate** returns **true**

Combination



	Map/OrderedMap	List	Set/OrderedSet	Stack	Seq
concat	X	X	X	X	X
flatten	X	X	X	X	X
flatMap	X	X	X	X	X
interpose		X		X	
interleave		X		X	
splice		X		X	
zip		X		X	
zipWith		X		X	

Reducing



- All these methods are available on all collection types
- `reduce(reducer, initialValue)` - reduces collection to a single value by calling `reducer` with latest value and an element from collection; `reducer` returns next value
- `reduceRight(reducer, initialValue)` - same as `reduce`, but elements are passed to `reducer` in reverse order
- `every(predicate)` - returns boolean indicating whether `predicate` returns true for every element
- `some(predicate)` - returns boolean indicating whether `predicate` returns true for any element
- `join(separator = ',')` - returns string formed by concatenating the `toString` value of all elements with `separator` string between them
- `isEmpty()` - returns boolean indicating whether collection is empty
- `count(predicate)` - returns count of elements where `predicate` returns `true` or count of all elements if predicate is omitted
- `countBy(grouper)` - returns a `KeyedSeq` where keys are ? and values are `Iterables` over elements in the same group; `grouper` is passed each element and returns its group

Search for Value



	Map/OrderedMap	List	Set/OrderedSet	Stack	Seq
find	X	X	X	X	X
findLast	X	X	X	X	X
findEntry	X	X	X	X	X
findLastEntry	X	X	X	X	X
max	X	X	X	X	X
maxBy	X	X	X	X	X
min	X	X	X	X	X
minBy	X	X	X	X	X
keyOf	X				
lastKeyOf	X				
findKey	X				
findLastKey	X				
indexOf		X		X	
lastIndexOf		X		X	
findIndex		X		X	
findLastIndex		X		X	

Comparison



- All collection types support these methods
 - `isSubset`
 - `isSuperset`

Sequence Functions



	Map/OrderedMap	List	Set/OrderedSet	Stack	Seq
flip	X				
mapKeys	X				
mapEntries	X				

Summary

- Immutability has **many benefits** and **few drawbacks**
- **Persistent data structures** are an important feature
 - avoid immutability libraries that don't implement these
- **Immutable** is a great library!
 - learning curve is primarily due to size of API
 - each piece is relatively simple to learn

The End

- Thanks so much for attending my talk!
- Feel free to find me later and ask questions about immutability or anything in the JavaScript world
- Check out my **talk on React tomorrow** at **2 PM** in **room A2**

- **Contact me**

Mark Volkmann, Object Computing, Inc.

Email: mark@ociweb.com

Twitter: [@mark_volkman](https://twitter.com/mark_volkman)

GitHub: [mvolkmann](https://github.com/mvolkmann)

Website: <http://ociweb.com/mark>