cssinjs

A brownbag Q&A at

MX®

by Seth House

@whiteinge
seth@eseth.com

A (little) history

Glamor vs <insert lib here>

Glamor vs <insert lib here>

- Discussed multiple times in the last few years.

Glamor vs <insert lib here>

- Discussed multiple times in the last few years.
- Teams have "CSS rewrite" fatigue.

Glamor vs <insert lib here>

- Discussed multiple times in the last few years.
- Teams have "CSS rewrite" fatigue.
- Rewriting CSS across all our FE projects is an *enormous* undertaking.

Glamor vs <insert lib here>

- Discussed multiple times in the last few years.
- Teams have "CSS rewrite" fatigue.
- Rewriting CSS across all our FE projects is an *enormous* undertaking.
- Outlier CSS implementations:

Glamor vs <insert lib here>

- Discussed multiple times in the last few years.
- Teams have "CSS rewrite" fatigue.
- Rewriting CSS across all our FE projects is an *enormous* undertaking.
- Outlier CSS implementations:
    - Increase friction for cross-team "developer borrowing",

Glamor vs <insert lib here>

- Discussed multiple times in the last few years.
- Teams have "CSS rewrite" fatigue.
- Rewriting CSS across all our FE projects is an *enormous* undertaking.
- Outlier CSS implementations:
    - Increase friction for cross-team "developer borrowing",
    - and for cross-team developer transfers;

Glamor vs <insert lib here>

- Discussed multiple times in the last few years.
- Teams have "CSS rewrite" fatigue.
- Rewriting CSS across all our FE projects is an *enormous* undertaking.
- Outlier CSS implementations:
  - Increase friction for cross-team "developer borrowing",
  - and for cross-team developer transfers;
  - Increase technical debt across MX.

Why Glamor?

- (Very, very) Simple API:
  `const className = css({ })`

Why Glamor?

- (Very, very) Simple API:
  `const className = css({ })`
- No build-time or editor tooling needed.

Why Glamor?

- (Very, very) Simple API:
  `const className = css({ })`
- No build-time or editor tooling needed.
- Common API with a handful of other libs.

Why Glamor?

- (Very, very) Simple API:
  `const className = css({ })`
- No build-time or editor tooling needed.
- Common API with a handful of other libs.
- Very close to writing vanilla CSS (but with namespacing).

Why Glamor?

- (Very, very) Simple API:
  `const className = css({ })`
- No build-time or editor tooling needed.
- Common API with a handful of other libs.
- Very close to writing vanilla CSS (but with namespacing).

Why not Glamor?

Why Glamor?

- (Very, very) Simple API:
  `const className = css({ })`
- No build-time or editor tooling needed.
- Common API with a handful of other libs.
- Very close to writing vanilla CSS (but with namespacing).

Why not Glamor?

- Unmaintained.

Why Glamor?

- (Very, very) Simple API:
  `const className = css({ })`
- No build-time or editor tooling needed.
- Common API with a handful of other libs.
- Very close to writing vanilla CSS (but with namespacing).

Why not Glamor?

- Unmaintained.
- Large number of deps were a problem.

Why Glamor?

- (Very, very) Simple API:
  `const className = css({ })`
- No build-time or editor tooling needed.
- Common API with a handful of other libs.
- Very close to writing vanilla CSS (but with namespacing).

Why not Glamor?

- Unmaintained.
- Large number of deps were a problem.
- Some API aspects were something we wanted to avoid using.

Why Glamor?

- (Very, very) Simple API:
  `const className = css({ })`
- No build-time or editor tooling needed.
- Common API with a handful of other libs.
- Very close to writing vanilla CSS (but with namespacing).

Why not Glamor?

- Unmaintained.
- Large number of deps were a problem.
- Some API aspects were something we wanted to avoid using.
- Payload size could be improved.

Why Glamor?

- (Very, very) Simple API:
  `const className = css({ })`
- No build-time or editor tooling needed.
- Common API with a handful of other libs.
- Very close to writing vanilla CSS (but with namespacing).

Why not Glamor?

- Unmaintained.
- Large number of deps were a problem.
- Some API aspects were something we wanted to avoid using.
- Payload size could be improved.
- Styles bleed "downward" (namespacing is an imperfect encapsulation).

So we're stuck with the Glamor API?

So we're stuck with the Glamor API?

No. MX encourages exploration.

So we're stuck with the Glamor API?

No. MX encourages exploration.

- Data driven!

So we're stuck with the Glamor API?

No. MX encourages exploration.

- Data driven!
- Write a proposal (RFC).

So we're stuck with the Glamor API?

No. MX encourages exploration.

- Data driven!
- Write a proposal (RFC).
- Define *objective* success criteria and a timeline.

So we're stuck with the Glamor API?

No. MX encourages exploration.

- Data driven!
- Write a proposal (RFC).
- Define *objective* success criteria and a timeline. (Subjective proposals should be rare and have overwhelming cross-team support.)

So we're stuck with the Glamor API?

No. MX encourages exploration.

- Data driven!
- Write a proposal (RFC).
- Define *objective* success criteria and a timeline. (Subjective proposals should be rare and have overwhelming cross-team support.)
- Once the proposal is greenlit, run the experiment for the allotted time.

So we're stuck with the Glamor API?

No. MX encourages exploration.

- Data driven!
- Write a proposal (RFC).
- Define *objective* success criteria and a timeline. (Subjective proposals should be rare and have overwhelming cross-team support.)
- Once the proposal is greenlit, run the experiment for the allotted time.
- Evalutate the results according to the success criteria.

So we're stuck with the Glamor API?

No. MX encourages exploration.

- Data driven!
- Write a proposal (RFC).
- Define *objective* success criteria and a timeline. (Subjective proposals should be rare and have overwhelming cross-team support.)
- Once the proposal is greenlit, run the experiment for the allotted time.
- Evalutate the results according to the success criteria.
- Teams at MX will start to adopt the accepted new tech.

So we're stuck with the Glamor API?

No. MX encourages exploration.

- Data driven!
- Write a proposal (RFC).
- Define *objective* success criteria and a timeline. (Subjective proposals should be rare and have overwhelming cross-team support.)
- Once the proposal is greenlit, run the experiment for the allotted time.
- Evalutate the results according to the success criteria.
- Teams at MX will start to adopt the accepted new tech.

My guess: since we've dropped IE11 support our next pivot will be "the platform".
CSS variables for Kyper tokens, Shadow DOM for encapsulation.

cssinjs

https://github.com/mxenabled/cssinjs/

A wrapper library

- Not *Yet Another CSS-in-JS Library*.

A wrapper library

- Not *Yet Another CSS-in-JS Library*.
- ~45 SLOC wrapper around Free-Style,

A wrapper library

- Not *Yet Another CSS-in-JS Library*.
- ~45 SLOC wrapper around Free-Style,
- ...which is a ~300 SLOC library with zero dependencies.

A wrapper library

- Not *Yet Another CSS-in-JS Library*.
- ~45 SLOC wrapper around Free-Style,
- ...which is a ~300 SLOC library with zero dependencies.
- Goal: zero maintenance, zero fuss.

A wrapper library

- Not *Yet Another CSS-in-JS Library*.
- ~45 SLOC wrapper around Free-Style,
- ...which is a ~300 SLOC library with zero dependencies.
- Goal: zero maintenance, zero fuss.
- Glamor API-compatible (excepting the following caveats).

A wrapper library

- Not *Yet Another CSS-in-JS Library*.
- ~45 SLOC wrapper around Free-Style,
- ...which is a ~300 SLOC library with zero dependencies.
- Goal: zero maintenance, zero fuss.
- Glamor API-compatible (excepting the following caveats).
- Please read the `README`

A wrapper library

- Not *Yet Another CSS-in-JS Library*.
- ~45 SLOC wrapper around Free-Style,
- ...which is a ~300 SLOC library with zero dependencies.
- Goal: zero maintenance, zero fuss.
- Glamor API-compatible (excepting the following caveats).
- Please read the `README` (please).

Thank you!

- Kyper Team.

Thank you!

- Kyper Team.
- Jared Gerlach.

Thank you!

- Kyper Team.
- Jared Gerlach.
- David Alger.

Thank you!

- Kyper Team.
- Jared Gerlach.
- David Alger.
- Peter, Maddie, Sam, & several others.

Caveat: order-dependent styles

Caveat: order-dependent styles

```
// Doesn't work:
css({
  borderTop: '1px solid',
  borderColor: 'blue',
})

// Works (but still not a good idea):
css({
  borderRight: '1px solid',
  borderColor: 'blue',
})
```

Why?

Interlude:

Data Structures

Data structures

(a,b)-tree, 2-3 heap, 2-3 tree, 2-3-4 tree, AA tree, AF-heap, AList, AVL tree, Abstract syntax tree, Adaptive k-d tree, Adjacency list, Adjacency matrix, Alternating decision tree, And-inverter graph, And-or tree, Array, Array list, Association list, Associative array, B sharp tree, B*-tree, B+ tree, B-heap, B-tree, BK-tree, BSP tree, Beap, Bin, Binary decision diagram, Binary heap, Binary search tree, Binary tree, Binomial heap, Bit array, Bit field, Bitboard, Bitmap, Bloom filter, Bounding interval hierarchy, Bounding volume hierarchy, Brodal queue, Bx-tree, C tree, Cartesian tree, Circular buffer, Collection, Compressed suffix array, Conc-tree list, Container, Control table, Count-min sketch, Cover tree, Ctrie, D-ary heap, Dancing tree, Decision tree, Difference list, Directed acyclic graph, Directed acyclic word graph, Directed graph, Disjoint-set, Disjoint-set data structure (Union-find data structure), Distributed hash table, Dope vector, Double hashing, Double-ended priority queue, Double-ended queue, Doubly connected edge list also known as half-edge, Doubly linked list, Dynamic array, Dynamic perfect hash table, Enfilade, Expectiminimax tree, Exponential tree, Expression tree, FM-index, Fenwick tree, Fibonacci heap, Finger tree, Free list, Fusion tree, Gap buffer, Generalised suffix tree, Graph, Graph-structured stack, Hash array mapped trie, Hash list, Hash table, Hash tree, Hash trie, Hashed array tree, Heap, Hilbert R-tree, Hypergraph, Iliffe vector, Image, Implicit k-d tree, Interval tree, Judy array, K-ary tree, K-d tree, Koorde, Left-child right-sibling binary tree, Leftist heap, Leonardo heap, Lexicographic Search Tree, Lightmap, Linear octree, Link/cut tree, Linked list, List, Log-structured merge-tree, Lookup table, M-tree, Map, Matrix, Merkle tree, Metric tree, Min/max k-d tree, MinHash, Minimax tree, Multigraph, Multimap, Multiset, Octree, Order statistic tree, Pagoda, Pairing heap, Parallel array, Parse tree, Piece table, Prefix hash tree, Priority queue, Propositional directed acyclic graph, Quad-edge, Quadtree, Queap, Queue, Quotient filter, R* tree, R+ tree, R-tree, R* tree, R+ tree, Radix tree, Randomized binary search tree, Range tree, Rapidly exploring random tree, Record, Red-black tree, Relaxed k-d tree, Retrieval Data Structure, Rolling hash, Rope, Rose tree, Routing table, SPQR-tree, Scapegoat tree, Scene graph, Segment tree, Self-balancing binary search tree, Self-balancing tree, Self-organizing list, Set, Skew heap, Skip list, Soft heap, Sorted array, Spaghetti stack, Sparse matrix, Splay tree, Stack, Suffix array, Suffix tree, Symbol table, T-tree, Tagged union, Tango tree, Ternary heap, Ternary tree, Threaded binary tree, Top tree, Treap, Tree, Trie, Tuple, UB-tree, Union, Unrolled linked list, VList, VP-tree, Van Emde Boas tree, Variable-length array, WAVL tree, Weak heap, Weight-balanced tree, Winged edge, X-fast trie, X-tree, XOR linked list, Xor linked list, Y-fast trie, Z-order, Zero-suppressed decision diagram, Zipper, bag, discriminated union, disjoint union, structure, variant, variant record.

Associative array

Associative array,

Associative array

Associative array, dictionary,

Associative array

Associative array, dictionary, hash table,

Associative array

Associative array, dictionary, hash table, hash map,

Associative array

Associative array, dictionary, hash table, hash map, table (etc).

Associative array

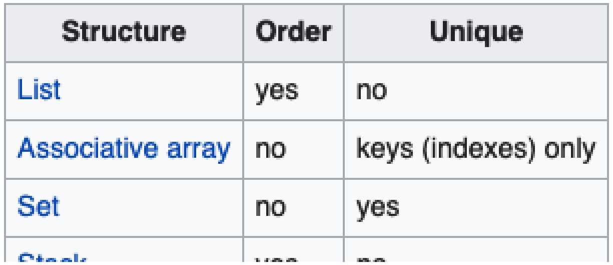
Associative array, dictionary, hash table, hash map, table (etc).

```
const MyObject = {
    foo: 'Foo!',
    bar: 'Bar!',
}
```

Associative array

Associative array, dictionary, hash table, hash map, table (etc).

```
const MyObject = {
    foo: 'Foo!',
    bar: 'Bar!',
}
```

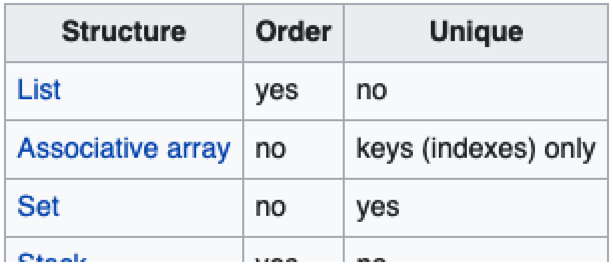


Structure	Order	Unique
List	yes	no
Associative array	no	keys (indexes) only
Set	no	yes

Associative array

Associative array, dictionary, hash table, hash map, table (etc).

```
const MyObject = {
    foo: 'Foo!',
    bar: 'Bar!',
}
```



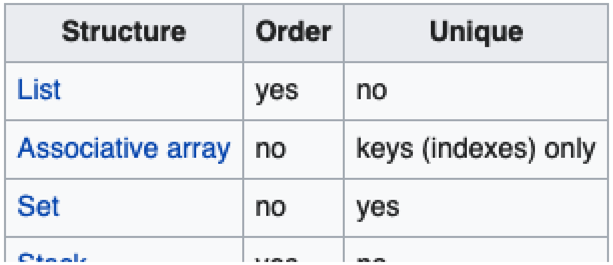
Structure	Order	Unique
List	yes	no
Associative array	no	keys (indexes) only
Set	no	yes

Optimized for fast *lookup*. Not traversal, not insertion order, not search, etc.

Associative array

Associative array, dictionary, hash table, hash map, table (etc).

```
const MyObject = {
    foo: 'Foo!',
    bar: 'Bar!',
}
```



Structure	Order	Unique
List	yes	no
Associative array	no	keys (indexes) only
Set	no	yes
Stack	yes	no

Optimized for fast *lookup*. Not traversal, not insertion order, not search, etc.

JavaScript, Python, Ruby, C#, Java, Lua, others -- all *unordered*! *

Explore data structure characteristics

```
console.log('Generating big data structures...')
var uniqueId = () => (Math.random() + 1).toString(36).substring(2)
var BigArray = Array.from({length: 1000000}, () => uniqueId())
var BigObject = Object.fromEntries(BigArray.map((x, i) => [x, i]))
var firstItem = BigArray[0]
var lastItem = BigArray[BigArray.length - 1]
console.log('...done. Starting lookups:')

console.time('Array traversal (first)')
BigArray.find(x => x === firstItem)
console.timeEnd('Array traversal (first)')

console.time('Array traversal (last)')
BigArray.find(x => x === lastItem)
console.timeEnd('Array traversal (last)')

console.time('Object lookup (first)')
BigObject[firstItem]
console.timeEnd('Object lookup (first)')

console.time('Object lookup (last)')
BigObject[lastItem]
console.timeEnd('Object lookup (last)')
```

Insertion-order-preserving structure

```
const MyMap = new Map();
MyMap.set('foo', 'Foo!');
```

Insertion-order-preserving structure

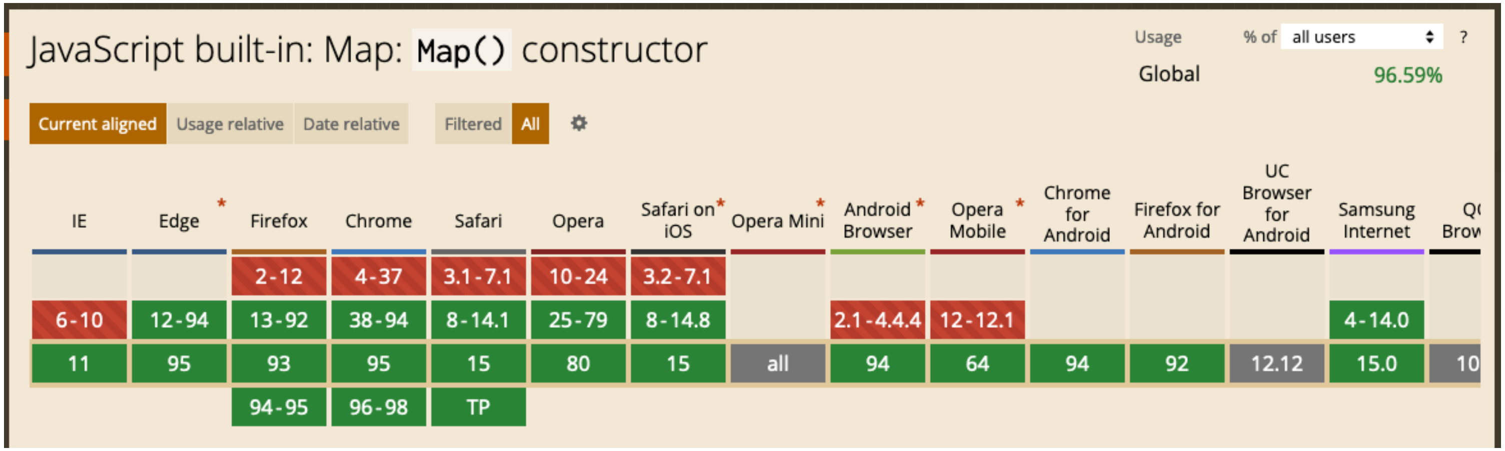
```
const MyMap = new Map();
MyMap.set('foo', 'Foo!');
```

> The keys in Map are ordered in a simple, straightforward way: A Map object iterates entries, keys, and values in the order of entry insertion.
>
> -- MDN

Insertion-order-preserving structure

```
const MyMap = new Map();
MyMap.set('foo', 'Foo!');
```

> The keys in Map are ordered in a simple, straightforward way: A Map object iterates entries, keys, and values in the order of entry insertion.
>
> -- MDN



JavaScript built-in: Map: `Map()` constructor

Usage % of all users ?

Global 96.59%

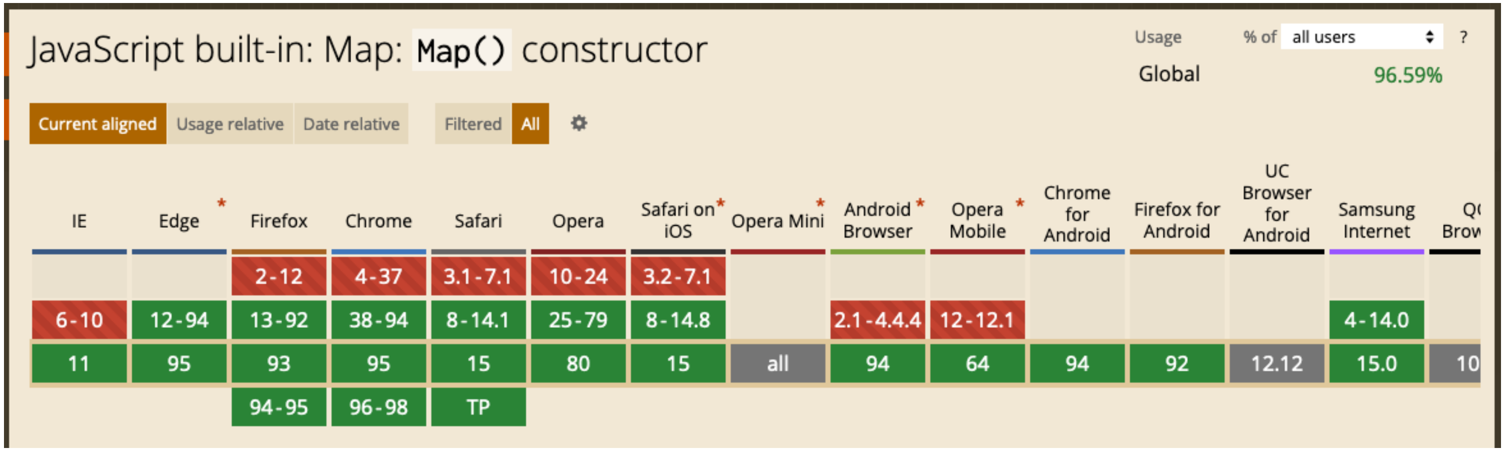
Current aligned | Usage relative | Date relative | Filtered | All

IE	Edge *	Firefox	Chrome	Safari	Opera	Safari on iOS *	Opera Mini *	Android Browser *	Opera Mobile *	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	Q Brow
		2-12	4-37	3.1-7.1	10-24	3.2-7.1								
6-10	12-94	13-92	38-94	8-14.1	25-79	8-14.8		2.1-4.4.4	12-12.1				4-14.0	
11	95	93	95	15	80	15	all	94	64	94	92	12.12	15.0	10
		94-95	96-98	TP										

Insertion-order-preserving structure

```javascript
const MyMap = new Map();
MyMap.set('foo', 'Foo!');
```

> The keys in Map are ordered in a simple, straightforward way: A Map object iterates entries, keys, and values in the order of entry insertion.
>
> -- MDN



JavaScript built-in: Map: `Map()` constructor

Usage: Global — % of all users: 96.59%

Current aligned | Usage relative | Date relative | Filtered | All

IE	Edge *	Firefox	Chrome	Safari	Opera	Safari on iOS *	Opera Mini *	Android Browser *	Opera Mobile *	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	Q Brow
		2-12	4-37	3.1-7.1	10-24	3.2-7.1								
6-10	12-94	13-92	38-94	8-14.1	25-79	8-14.8		2.1-4.4.4	12-12.1				4-14.0	
11	95	93	95	15	80	15	all	94	64	94	92	12.12	15.0	10
		94-95	96-98	TP										

Python, Ruby, C#, Java, Lua, and others also have separate order-preserving structures.

* Well actually...

* Well actually...

- JS added preservation of insertion order to *some* `Object` internals to the ES5 spec, then more in ES2015, and more in ES2020.

* Well actually...

- JS added preservation of insertion order to *some* `Object` internals to the ES5 spec, then more in ES2015, and more in ES2020.
- Common JS engines have largely implemented those specifications now.

* Well actually...

- JS added preservation of insertion order to *some* `Object` internals to the ES5 spec, then more in ES2015, and more in ES2020.
- Common JS engines have largely implemented those specifications now.
- Python and Ruby have recently(!) made similar order-preserving changes to the base data structure.

* Well actually...

- JS added preservation of insertion order to *some* `Object` internals to the ES5 spec, then more in ES2015, and more in ES2020.
- Common JS engines have largely implemented those specifications now.
- Python and Ruby have recently(!) made similar order-preserving changes to the base data structure.
- Is all this advice now antiquated?

Caveat emptor

Caveat emptor

- Unordered associative arrays have a long history and change happens slowly.

Caveat emptor

- Unordered associative arrays have a long history and change happens slowly.
- Are JavaScript, Python, & Ruby setting a new long-term trend? Or will they be outliers?

Caveat emptor

- Unordered associative arrays have a long history and change happens slowly.
- Are JavaScript, Python, & Ruby setting a new long-term trend? Or will they be outliers?
- May be confusing if you find yourself in another language in the future.

Caveat emptor

- Unordered associative arrays have a long history and change happens slowly.
- Are JavaScript, Python, & Ruby setting a new long-term trend? Or will they be outliers?
- May be confusing if you find yourself in another language in the future.
- Order preservation may not survive serialization.

  (Requires order-preserving support in both the source and target languages, plus any intermediaries, plus the serializer/deserializer implementations. Semantics may (will!) differ beteween serialization formats (JSON, XML, protobuf, etc).)

Caveat emptor

- Unordered associative arrays have a long history and change happens slowly.
- Are JavaScript, Python, & Ruby setting a new long-term trend? Or will they be outliers?
- May be confusing if you find yourself in another language in the future.
- Order preservation may not survive serialization.
  (Requires order-preserving support in both the source and target languages, plus any intermediaries, plus the serializer/deserializer implementations. Semantics may (will!) differ beteween serialization formats (JSON, XML, protobuf, etc).)
- Additional interoperability concerns on unusual or niche JS engines.
  (Embedded devices, games, inside other programs (Excel, PDFs, etc).)

Caveat emptor

- Unordered associative arrays have a long history and change happens slowly.
- Are JavaScript, Python, & Ruby setting a new long-term trend? Or will they be outliers?
- May be confusing if you find yourself in another language in the future.
- Order preservation may not survive serialization.

  (Requires order-preserving support in both the source and target languages, plus any intermediaries, plus the serializer/deserializer implementations. Semantics may (will!) differ beteween serialization formats (JSON, XML, protobuf, etc).)
- Additional interoperability concerns on unusual or niche JS engines.

  (Embedded devices, games, inside other programs (Excel, PDFs, etc).)
- Third-party libraries may target different browsers than you do and handle ordering explicitly.

  (E.g. the CSS library that we are currently using, and React (as a console warning).)

Caveat emptor

- Unordered associative arrays have a long history and change happens slowly.
- Are JavaScript, Python, & Ruby setting a new long-term trend? Or will they be outliers?
- May be confusing if you find yourself in another language in the future.
- Order preservation may not survive serialization.
  (Requires order-preserving support in both the source and target languages, plus any intermediaries, plus the serializer/deserializer implementations. Semantics may (will!) differ beteween serialization formats (JSON, XML, protobuf, etc).)
- Additional interoperability concerns on unusual or niche JS engines.
  (Embedded devices, games, inside other programs (Excel, PDFs, etc).)
- Third-party libraries may target different browsers than you do and handle ordering explicitly.
  (E.g. the CSS library that we are currently using, and React (as a console warning).)
- We have usable order-preserving-specific data structures.

Interlude Concluded

Caveat: order-dependent styles

```
// Bad:
css({
  borderTop: '1px solid',
  borderColor: 'blue',
})

// Good:
css({
  borderRightWidth: '1px',
  borderRightStyle: 'solid',
  borderColor: 'blue',
})
```

Caveat: order-dependent styles

```
// Bad:
css({
  borderTop: '1px solid',
  borderColor: 'blue',
})

// Good:
css({
  borderRightWidth: '1px',
  borderRightStyle: 'solid',
  borderColor: 'blue',
})
```

Free-Style sorts object keys lexicographically (alphabetically) in order to produce a deterministic hash across many JS environments.

Caveat: order-dependent styles

```
// Bad:
css({
  borderTop: '1px solid',
  borderColor: 'blue',
})

// Good:
css({
  borderRightWidth: '1px',
  borderRightStyle: 'solid',
  borderColor: 'blue',
})
```

Free-Style sorts object keys lexicographically (alphabetically) in order to produce a deterministic hash across many JS environments.

Feel free to mix-and-match longhand and shorthand CSS styles! However, beware order-dependent styles.

Aside: inline styles too

```
<div style={{
    borderTop: '1px solid',
    borderColor: 'blue',
  }}>
    Hello, world.
</div>
```

Aside: inline styles too

```
<div style={{
    borderTop: '1px solid',
    borderColor: 'blue',
  }}>
    Hello, world.
</div>
```

React 16.13.0+ detects potentially conflicting styles and throws a warning.