You Don't Know JS Yet: ES.Next & Beyond - 2nd Edition

Chapter 5: Collections

NOTE:
Work in progress

.

.

.

.

.

.

.

NOTE:
Everything below here is previous text from 1st edition, and is only here for reference while 2nd edition work is underway. **Please ignore this stuff.**

Structured collection and access to data is a critical component of just about any JS program. From the beginning of the language up to this point, the array and the object have been our primary mechanism for creating data structures. Of course, many higher-level data structures have been built on top of these, as user-land libraries.

As of ES6, some of the most useful (and performance-optimizing!) data structure abstractions have been added as native components of the language.

We'll start this chapter first by looking at *TypedArrays*, technically contemporary to ES5 efforts several years ago, but only standardized as companions to WebGL and not JavaScript itself. As of ES6, these have been adopted directly by the language specification, which gives them first-class status.

Maps are like objects (key/value pairs), but instead of just a string for the key, you can use any value -- even another object or map! Sets are similar to arrays (lists of values), but the values are unique; if you add a duplicate, it's ignored. There are also weak (in relation to memory/garbage collection) counterparts: WeakMap and WeakSet.

TypedArrays

As we cover in the *Types & Grammar* title of this series, JS does have a set of built-in types, like `number` and `string`. It'd be tempting to look at a feature named "typed array" and assume it means an array of a specific type of values, like an array of only strings.

However, typed arrays are really more about providing structured access to binary data using array-like semantics (indexed access, etc.). The "type" in the name refers to a "view" layered on type of the bucket of bits, which is essentially a mapping of whether the bits should be viewed as an array of 8-bit signed integers, 16-bit signed integers, and so on.

How do you construct such a bit-bucket? It's called a "buffer," and you construct it most directly with the `ArrayBuffer(..)` constructor:

```
var buf = new ArrayBuffer( 32 );
buf.byteLength;                         // 32
```

`buf` is now a binary buffer that is 32-bytes long (256-bits), that's pre-initialized to all `0`s. A buffer by itself doesn't really allow you any interaction except for checking its `byteLength` property.

Tip: Several web platform features use or return array buffers, such as `FileReader#readAsArrayBuffer(..)`, `XMLHttpRequest#send(..)`, and `ImageData` (canvas data).

But on top of this array buffer, you can then layer a "view," which comes in the form of a typed array. Consider:

```
var arr = new Uint16Array( buf );
arr.length;                             // 16
```

`arr` is a typed array of 16-bit unsigned integers mapped over the 256-bit `buf` buffer, meaning you get 16 elements.

Endianness

It's very important to understand that the `arr` is mapped using the endian-setting (big-endian or little-endian) of the platform the JS is running on. This can be an issue if the binary data is created with one endianness but interpreted on a platform with the opposite endianness.

Endian means if the low-order byte (collection of 8-bits) of a multi-byte number -- such as the 16-bit unsigned ints we created in the earlier snippet -- is on the right or the left of the number's bytes.

For example, let's imagine the base-10 number `3085`, which takes 16-bits to represent. If you have just one 16-bit number container, it'd be represented in binary as `0000110000001101` (hexadecimal `0c0d`) regardless of endianness.

But if `3085` was represented with two 8-bit numbers, the endianness would significantly affect its storage in memory:

- `0000110000001101` / `0c0d` (big endian)
- `0000110100001100` / `0d0c` (little endian)

If you received the bits of `3085` as `0000110100001100` from a little-endian system, but you layered a view on top of it in a big-endian system, you'd instead see value `3340` (base-10) and `0d0c` (base-16).

Little endian is the most common representation on the web these days, but there are definitely browsers where that's not true. It's important that you understand the endianness of both the producer and consumer of a chunk of binary data.

From MDN, here's a quick way to test the endianness of your JavaScript:

```
var littleEndian = (function() {
    var buffer = new ArrayBuffer( 2 );
    new DataView( buffer ).setInt16( 0, 256, true );
    return new Int16Array( buffer )[0] === 256;
})();
```

`littleEndian` will be `true` or `false`; for most browsers, it should return `true`. This test uses `DataView(..)`, which allows more low-level, fine-grained control over accessing (setting/getting) the bits from the view you layer over the buffer. The third parameter of the `setInt16(..)` method in the previous snippet is for telling the `DataView` what endianness you're wanting it to use for that operation.

Warning: Do not confuse endianness of underlying binary storage in array buffers with how a given number is represented when exposed in a JS program. For example, `(3085).toString(2)` returns `"110000001101"`, which with an assumed leading four `"0"`s appears to be the big-endian representation. In fact, this representation is based on a single 16-bit view, not a view of two 8-bit bytes. The `DataView` test above is the best way to determine endianness for your JS environment.

Multiple Views

A single buffer can have multiple views attached to it, such as:

```
var buf = new ArrayBuffer( 2 );

var view8 = new Uint8Array( buf );
var view16 = new Uint16Array( buf );

view16[0] = 3085;
view8[0];                       // 13
view8[1];                       // 12

view8[0].toString( 16 );        // "d"
view8[1].toString( 16 );        // "c"

// swap (as if endian!)
var tmp = view8[0];
view8[0] = view8[1];
view8[1] = tmp;

view16[0];                      // 3340
```

The typed array constructors have multiple signature variations. We've shown so far only passing them an existing buffer. However, that form also takes two extra parameters: `byteOffset` and `length`. In other words, you can start the typed array view at a location other than `0` and you can make it span less than the full length of the buffer.

If the buffer of binary data includes data in non-uniform size/location, this technique can be quite useful.

For example, consider a binary buffer that has a 2-byte number (aka "word") at the beginning, followed by two 1-byte numbers, followed by a 32-bit floating point number. Here's how you can access that data with multiple views on the same buffer, offsets, and lengths:

```
var first = new Uint16Array( buf, 0, 2 )[0],
    second = new Uint8Array( buf, 2, 1 )[0],
    third = new Uint8Array( buf, 3, 1 )[0],
    fourth = new Float32Array( buf, 4, 4 )[0];
```

TypedArray Constructors

In addition to the `(buffer,[offset, [length]])` form examined in the previous section, typed array constructors also support these forms:

- [constructor] `(length)`: Creates a new view over a new buffer of `length` bytes
- [constructor] `(typedArr)`: Creates a new view and buffer, and copies the contents from the `typedArr` view
- [constructor] `(obj)`: Creates a new view and buffer, and iterates over the array-like or object `obj` to copy its contents

The following typed array constructors are available as of ES6:

- `Int8Array` (8-bit signed integers), `Uint8Array` (8-bit unsigned integers)
  - `Uint8ClampedArray` (8-bit unsigned integers, each value clamped on setting to the `0` - `255` range)
- `Int16Array` (16-bit signed integers), `Uint16Array` (16-bit unsigned integers)
- `Int32Array` (32-bit signed integers), `Uint32Array` (32-bit unsigned integers)
- `Float32Array` (32-bit floating point, IEEE-754)
- `Float64Array` (64-bit floating point, IEEE-754)

Instances of typed array constructors are almost the same as regular native arrays. Some differences include having a fixed length and the values all being of the same "type."

However, they share most of the same `prototype` methods. As such, you likely will be able to use them as regular arrays without needing to convert.

For example:

```
var a = new Int32Array( 3 );
a[0] = 10;
a[1] = 20;
a[2] = 30;

a.map( function(v){
    console.log( v );
} );
// 10 20 30

a.join( "-" );
// "10-20-30"
```

Warning: You can't use certain `Array.prototype` methods with TypedArrays that don't make sense, such as the mutators (`splice(..)` , `push(..)` , etc.) and `concat(..)` .

Be aware that the elements in TypedArrays really are constrained to the declared bit sizes. If you have a `Uint8Array` and try to assign something larger than an 8-bit value into one of its elements, the value wraps around so as to stay within the bit length.

This could cause problems if you were trying to, for instance, square all the values in a TypedArray. Consider:

```
var a = new Uint8Array( 3 );
a[0] = 10;
a[1] = 20;
a[2] = 30;

var b = a.map( function(v){
    return v * v;
} );

b;              // [100, 144, 132]
```

The `20` and `30` values, when squared, resulted in bit overflow. To get around such a limitation, you can use the `TypedArray#from(..)` function:

```
var a = new Uint8Array( 3 );
a[0] = 10;
a[1] = 20;
a[2] = 30;

var b = Uint16Array.from( a, function(v){
    return v * v;
} );

b;              // [100, 400, 900]
```

See the " `Array.from(..)` Static Function" section in Chapter 6 for more information about the `Array.from(..)` that is shared with TypedArrays. Specifically, the "Mapping" section explains the mapping

function accepted as its second argument.

One interesting behavior to consider is that TypedArrays have a `sort(..)` method much like regular arrays, but this one defaults to numeric sort comparisons instead of coercing values to strings for lexicographic comparison. For example:

```
var a = [ 10, 1, 2, ];
a.sort();                               // [1,10,2]

var b = new Uint8Array( [ 10, 1, 2 ] );
b.sort();                               // [1,2,10]
```

The `TypedArray#sort(..)` takes an optional compare function argument just like `Array#sort(..)`, which works in exactly the same way.

Maps

If you have a lot of JS experience, you know that objects are the primary mechanism for creating unordered key/value-pair data structures, otherwise known as maps. However, the major drawback with objects-as-maps is the inability to use a non-string value as the key.

For example, consider:

```
var m = {};

var x = { id: 1 },
    y = { id: 2 };

m[x] = "foo";
m[y] = "bar";

m[x];                           // "bar"
m[y];                           // "bar"
```

What's going on here? The two objects `x` and `y` both stringify to `"[object Object]"`, so only that one key is being set in `m`.

Some have implemented fake maps by maintaining a parallel array of non-string keys alongside an array of the values, such as:

```
var keys = [], vals = [];

var x = { id: 1 },
    y = { id: 2 };

keys.push( x );
vals.push( "foo" );

keys.push( y );
vals.push( "bar" );
```

```
keys[0] === x;                  // true
vals[0];                        // "foo"

keys[1] === y;                  // true
vals[1];                        // "bar"
```

Of course, you wouldn't want to manage those parallel arrays yourself, so you could define a data structure with methods that automatically do the management under the covers. Besides having to do that work yourself, the main drawback is that access is no longer O(1) time-complexity, but instead is O(n).

But as of ES6, there's no longer any need to do this! Just use `Map(..)`:

```
var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );
m.set( y, "bar" );

m.get( x );                     // "foo"
m.get( y );                     // "bar"
```

The only drawback is that you can't use the `[ ]` bracket access syntax for setting and retrieving values. But `get(..)` and `set(..)` work perfectly suitably instead.

To delete an element from a map, don't use the `delete` operator, but instead use the `delete(..)` method:

```
m.set( x, "foo" );
m.set( y, "bar" );

m.delete( y );
```

You can clear the entire map's contents with `clear()`. To get the length of a map (i.e., the number of keys), use the `size` property (not `length`):

```
m.set( x, "foo" );
m.set( y, "bar" );
m.size;                         // 2

m.clear();
m.size;                         // 0
```

The `Map(..)` constructor can also receive an iterable (see "Iterators" in Chapter 3), which must produce a list of arrays, where the first item in each array is the key and the second item is the value. This format for iteration is identical to that produced by the `entries()` method, explained in the next section. That makes it easy to make a copy of a map:

```
var m2 = new Map( m.entries() );

// same as:
var m2 = new Map( m );
```

Because a map instance is an iterable, and its default iterator is the same as `entries()`, the second shorter form is more preferable.

Of course, you can just manually specify an *entries* list (array of key/value arrays) in the `Map(..)` constructor form:

```
var x = { id: 1 },
    y = { id: 2 };

var m = new Map( [
    [ x, "foo" ],
    [ y, "bar" ]
] );

m.get( x );                     // "foo"
m.get( y );                     // "bar"
```

Map Values

To get the list of values from a map, use `values(..)`, which returns an iterator. In Chapters 2 and 3, we covered various ways to process an iterator sequentially (like an array), such as the `...` spread operator and the `for..of` loop. Also, "Arrays" in Chapter 6 covers the `Array.from(..)` method in detail. Consider:

```
var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );
m.set( y, "bar" );

var vals = [ ...m.values() ];

vals;                           // ["foo","bar"]
Array.from( m.values() );       // ["foo","bar"]
```

As discussed in the previous section, you can iterate over a map's entries using `entries()` (or the default map iterator). Consider:

```
var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );
```

```
m.set( y, "bar" );

var vals = [ ...m.entries() ];

vals[0][0] === x;               // true
vals[0][1];                     // "foo"

vals[1][0] === y;               // true
vals[1][1];                     // "bar"
```

Map Keys

To get the list of keys, use `keys()`, which returns an iterator over the keys in the map:

```
var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );
m.set( y, "bar" );

var keys = [ ...m.keys() ];

keys[0] === x;                  // true
keys[1] === y;                  // true
```

To determine if a map has a given key, use `has(..)`:

```
var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );

m.has( x );                     // true
m.has( y );                     // false
```

Maps essentially let you associate some extra piece of information (the value) with an object (the key) without actually putting that information on the object itself.

While you can use any kind of value as a key for a map, you typically will use objects, as strings and other primitives are already eligible as keys of normal objects. In other words, you'll probably want to continue to use normal objects for maps unless some or all of the keys need to be objects, in which case map is more appropriate.

Warning: If you use an object as a map key and that object is later discarded (all references unset) in attempt to have garbage collection (GC) reclaim its memory, the map itself will still retain its entry. You will need to remove the entry from the map for it to be GC-eligible. In the next section, we'll see WeakMaps as a better option for object keys and GC.

WeakMaps

WeakMaps are a variation on maps, which has most of the same external behavior but differs underneath in how the memory allocation (specifically its GC) works.

WeakMaps take (only) objects as keys. Those objects are held *weakly*, which means if the object itself is GC'd, the entry in the WeakMap is also removed. This isn't observable behavior, though, as the only way an object can be GC'd is if there's no more references to it -- once there are no more references to it, you have no object reference to check if it exists in the WeakMap.

Otherwise, the API for WeakMap is similar, though more limited:

```
var m = new WeakMap();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );

m.has( x );                     // true
m.has( y );                     // false
```

WeakMaps do not have a `size` property or `clear()` method, nor do they expose any iterators over their keys, values, or entries. So even if you unset the `x` reference, which will remove its entry from `m` upon GC, there is no way to tell. You'll just have to take JavaScript's word for it!

Just like Maps, WeakMaps let you soft-associate information with an object. But they are particularly useful if the object is not one you completely control, such as a DOM element. If the object you're using as a map key can be deleted and should be GC-eligible when it is, then a WeakMap is a more appropriate option.

It's important to note that a WeakMap only holds its *keys* weakly, not its values. Consider:

```
var m = new WeakMap();

var x = { id: 1 },
    y = { id: 2 },
    z = { id: 3 },
    w = { id: 4 };

m.set( x, y );

x = null;                       // { id: 1 } is GC-eligible
y = null;                       // { id: 2 } is GC-eligible
                                // only because { id: 1 } is

m.set( z, w );

w = null;                       // { id: 4 } is not GC-eligible
```

For this reason, WeakMaps are in my opinion better named "WeakKeyMaps."

Sets

A set is a collection of unique values (duplicates are ignored).

The API for a set is similar to map. The `add(..)` method takes the place of the `set(..)` method (somewhat ironically), and there is no `get(..)` method.

Consider:

```
var s = new Set();

var x = { id: 1 },
    y = { id: 2 };

s.add( x );
s.add( y );
s.add( x );

s.size;                         // 2

s.delete( y );
s.size;                         // 1

s.clear();
s.size;                         // 0
```

The `Set(..)` constructor form is similar to `Map(..)`, in that it can receive an iterable, like another set or simply an array of values. However, unlike how `Map(..)` expects *entries* list (array of key/value arrays), `Set(..)` expects a *values* list (array of values):

```
var x = { id: 1 },
    y = { id: 2 };

var s = new Set( [x,y] );
```

A set doesn't need a `get(..)` because you don't retrieve a value from a set, but rather test if it is present or not, using `has(..)`:

```
var s = new Set();

var x = { id: 1 },
    y = { id: 2 };

s.add( x );

s.has( x );                     // true
s.has( y );                     // false
```

Note: The comparison algorithm in `has(..)` is almost identical to `Object.is(..)` (see Chapter 6), except that `-0` and `0` are treated as the same rather than distinct.

Set Iterators

Sets have the same iterator methods as maps. Their behavior is different for sets, but symmetric with the behavior of map iterators. Consider:

```
var s = new Set();

var x = { id: 1 },
    y = { id: 2 };

s.add( x ).add( y );

var keys = [ ...s.keys() ],
    vals = [ ...s.values() ],
    entries = [ ...s.entries() ];

keys[0] === x;
keys[1] === y;

vals[0] === x;
vals[1] === y;

entries[0][0] === x;
entries[0][1] === x;
entries[1][0] === y;
entries[1][1] === y;
```

The `keys()` and `values()` iterators both yield a list of the unique values in the set. The `entries()` iterator yields a list of entry arrays, where both items of the array are the unique set value. The default iterator for a set is its `values()` iterator.

The inherent uniqueness of a set is its most useful trait. For example:

```
var s = new Set( [1,2,3,4,"1",2,4,"5"] ),
    uniques = [ ...s ];

uniques;                        // [1,2,3,4,"1","5"]
```

Set uniqueness does not allow coercion, so `1` and `"1"` are considered distinct values.

WeakSets

Whereas a WeakMap holds its keys weakly (but its values strongly), a WeakSet holds its values weakly (there aren't really keys).

```
var s = new WeakSet();

var x = { id: 1 },
    y = { id: 2 };

s.add( x );
```

```
s.add( y );

x = null;                       // `x` is GC-eligible
y = null;                       // `y` is GC-eligible
```

Warning: WeakSet values must be objects, not primitive values as is allowed with sets.

Review

ES6 defines a number of useful collections that make working with data in structured ways more efficient and effective.

TypedArrays provide "view"s of binary data buffers that align with various integer types, like 8-bit unsigned integers and 32-bit floats. The array access to binary data makes operations much easier to express and maintain, which enables you to more easily work with complex data like video, audio, canvas data, and so on.

Maps are key-value pairs where the key can be an object instead of just a string/primitive. Sets are unique lists of values (of any type).

WeakMaps are maps where the key (object) is weakly held, so that GC is free to collect the entry if it's the last reference to an object. WeakSets are sets where the value is weakly held, again so that GC can remove the entry if it's the last reference to that object.