Subclassing Arrays in ES2015

September 21, 2017

Prior to ES2015 (ES6), you couldn't really subclass an array in JavaScript without a few caveats, which kangax outlines in a fantastic post called How ECMAScript 5 still does not allow to subclass array. Now in ES2015, you can. This can be useful for defining custom collection classes that leverage all of the array methods and properties while also automatically implementing the iterable and iterator protocols, which I wrote about recently. In short, objects that implement these protocols can be looped through with the `for...of` loop or can be used with the spread operator.

Subclassing An Array With ES2015 Classes

Let's say you want to define a collection class with an `average` method. You can do that using a class:

```
class Collection extends Array {
  average(callback) {
    let total = this.reduce((total, item) => {
      return total + callback(item);
    }, 0);

    return total / this.length;
  }
}
```

You can then instantiate the class, passing in several items just like you would with an array, and see it work like a regular array with a custom `average` method:

Other Places You Can Find Me

5/30/2017 on eng.vdms.io
From Angular to Ember at Verizon Digital Media Services

5/24/2017 on sitepoint.com
Test-Driven Development in Node.js Video Course

1/12/2016 on codementor.io
JavaScript Testing Framework

```
const assert = require('assert');

let studentGrades = new Collection(
  { name: 'Leticia', grade: 95 },
  { name: 'Austen', grade: 85 },
  { name: 'Shane', grade: 90 }
);

assert.ok(studentGrades instanceof Array); // true
assert.ok(studentGrades instanceof Collection); // true
assert.strictEqual(studentGrades.constructor,
Collection); // true
assert.equal(studentGrades.length, 3); // true
studentGrades[3] = { name: 'Samantha', grade: 86 };
assert.equal(studentGrades.length, 4); // true
assert.equal(studentGrades.average(student =>
student.grade), 89); // true
studentGrades.push({ name: 'Leticia', grade: 95 });
assert.equal(studentGrades.length, 5); // true
studentGrades[10] = { name: 'Tom', grade: 75 };
assert.equal(studentGrades.length, 11); // true
studentGrades.length = 4;
assert.deepEqual(studentGrades, [
  { name: 'Leticia', grade: 95 },
  { name: 'Austen', grade: 85 },
  { name: 'Shane', grade: 90 },
  { name: 'Samantha', grade: 86 }
]); // true
```

Can You Subclass An Array Without A Class?

I then wondered if it was possible to subclass an array without a class. It turns out you can't, but you can read about several different approaches in How ECMAScript 5 still does not allow to subclass array. However, in that post kangax goes over an approach that does work, which uses something that is now standardized in ES2015, and that is `__proto__`. Basically this approach takes an array and changes its prototype to another object that inherits from `Array.prototype`. Here is a slightly modified version of that approach:

```
function Collection(...args) {
  Object.setPrototypeOf(args, Collection.prototype);
  return args;
}
Collection.prototype = Object.create(Array.prototype);
Collection.prototype.constructor = Collection;
Collection.prototype.average = function(callback) {
```

Comparison: Jasmine vs Mocha

11/26/2015 on codementor.io
Unit Testing & TDD in Node.js – Part 2

11/16/2015 on codementor.io
Unit Testing & TDD in Node.js – Part 1

10/30/2015 on codementor.io
Testing in Ember with Ember CLI

Subscribe

Email Address *

First Name

Last Name

Subscribe

```
  let total = this.reduce((total, item) => {
    return total + callback(item);
  }, 0);

  return total / this.length;
};
```

Here, the `Collection` constructor function returns an array instead of the object being constructed, and this array has its prototype changed to another object that inherits from `Array.prototype`.

This approach leverages Object.setPrototypeOf which was introduced in ES2015 as a way to set the prototype of an object to another object. Prior to this being introduced, the only way to change the prototype of an object was to use the non-standard `__proto__` property, so `Collection` would look like this instead:

```
function Collection(...args) {
  args.__proto__ = Collection.prototype;
  return args;
}
```

In ES2015, `Object.prototype.__proto__` was standardized, but it is recommended that you use `Object.getPrototypeOf` / `Reflect.getPrototypeOf` and `Object.setPrototypeOf` / `Reflect.setPrototypeOf`.

And all of the same assertions hold true:

```
const assert = require('assert');

let studentGrades = new Collection(
  { name: 'Leticia', grade: 95 },
  { name: 'Austen', grade: 85 },
  { name: 'Shane', grade: 90 }
);

assert.ok(studentGrades instanceof Array); // true
assert.ok(studentGrades instanceof Collection); // true
assert.strictEqual(studentGrades.constructor,
Collection); // true
assert.equal(studentGrades.length, 3); // true
```

```
studentGrades[3] = { name: 'Samantha', grade: 86 };
assert.equal(studentGrades.length, 4); // true
assert.equal(studentGrades.average(student =>
student.grade), 89); // true
studentGrades.push({ name: 'Leticia', grade: 95 });
assert.equal(studentGrades.length, 5); // true
studentGrades[10] = { name: 'Tom', grade: 75 };
assert.equal(studentGrades.length, 11); // true
studentGrades.length = 4;
assert.deepEqual(studentGrades, [
  { name: 'Leticia', grade: 95 },
  { name: 'Austen', grade: 85 },
  { name: 'Shane', grade: 90 },
  { name: 'Samantha', grade: 86 }
]); // true
```

Performance Comparison

If you checked out the documentation for Object.setPrototypeOf, you may have noticed the big red warning stating that changing the prototype of an object is a slow operation.

> **Warning:** Changing the `[[Prototype]]` of an object is, by the nature of how modern JavaScript engines optimize property accesses, a very slow operation, in ***every*** browser and JavaScript engine. The effects on performance of altering inheritance are subtle and far-flung, and are not limited to simply the time spent in `obj.__proto__ = ...` statement, but may extend to ***any*** code that has access to ***any*** object whose `[[Prototype]]` has been altered. If you care about performance you should avoid setting the `[[Prototype]]` of an object. Instead, create a new object with the desired `[[Prototype]]` using `Object.create()`.

I ran a JSPerf test comparing these two approaches to subclassing an array, and results varied across browsers.

Testing in Chrome 60.0.3112 / Mac OS X 10.12.6		
Test		**Ops/sec**
with a class	`new SubArray1(1, 2);`	3,147,413 ±0.93% 61% slower
without a class	`new SubArray2(1, 2);`	8,445,899 ±5.74% fastest

Testing in Safari 10.1.2 / Mac OS X 10.12.6		
Test		**Ops/sec**
with a class	`new SubArray1(1, 2);`	8,534,591 ±4.19% fastest
without a class	`new SubArray2(1, 2);`	2,205,772 ±5.24% 74% slower

Testing in Firefox 54.0.0 / Mac OS X 10.12.0		
Test		Ops/sec
with a class	`new SubArray1(1, 2);`	910,937 ±2.34% 35% slower
without a class	`new SubArray2(1, 2);`	1,394,007 ±2.54% fastest

Support

In terms of support, subclassing an array with a class isn't something that Babel can polyfill without a few caveats, since `__proto__` wasn't standardized until ES2015, and `Object.setPrototypeOf` wasn't added until ES2015. However, browser and Node support is pretty good, which you can see here.

Conclusion

To summarize, you can subclass an array using a class. Without a class, you can subclass an array by creating an array using the `Array` constructor or literal notation (`[]`) and changing its prototype to another object that inherits from `Array.prototype` . Although this approach is faster, I would still recommend using a class for better readability until performance becomes a problem.

Subscribe to keep up with my latest JavaScript and Ember adventures.

Disclaimer: Any viewpoints and opinions expressed in this article are those of David Tang and do not reflect those of my employer or any of my colleagues.
