You Don't Know JS: *this* & Object Prototypes

Appendix A: ES6 `class`

If there's any take-away message from the second half of this book (Chapters 4-6), it's that classes are an optional design pattern for code (not a necessary given), and that furthermore they are often quite awkward to implement in a `[[Prototype]]` language like JavaScript.

This awkwardness is *not* just about syntax, although that's a big part of it. Chapters 4 and 5 examined quite a bit of syntactic ugliness, from verbosity of `.prototype` references cluttering the code, to *explicit pseudo-polymorphism* (see Chapter 4) when you give methods the same name at different levels of the chain and try to implement a polymorphic reference from a lower-level method to a higher-level method. `.constructor` being wrongly interpreted as "was constructed by" and yet being unreliable for that definition is yet another syntactic ugly.

But the problems with class design are much deeper. Chapter 4 points out that classes in traditional class-oriented languages actually produce a *copy* action from parent to child to instance, whereas in `[[Prototype]]`, the action is **not** a copy, but rather the opposite -- a delegation link.

When compared to the simplicity of OLOO-style code and behavior delegation (see Chapter 6), which embrace `[[Prototype]]` rather than hide from it, classes stand out as a sore thumb in JS.

`class`

But we *don't* need to re-argue that case again. I re-mention those issues briefly only so that you keep them fresh in your mind now that we turn our attention to the ES6 `class` mechanism. We'll demonstrate here how it works, and look at whether or not `class` does anything substantial to address any of those "class" concerns.

Let's revisit the `Widget` / `Button` example from Chapter 6:

```
class Widget {
	constructor(width,height) {
		this.width = width || 50;
		this.height = height || 50;
		this.$elem = null;
	}
	render($where){
		if (this.$elem) {
			this.$elem.css( {
				width: this.width + "px",
				height: this.height + "px"
			} ).appendTo( $where );
		}
	}
}

class Button extends Widget {
	constructor(width,height,label) {
		super( width, height );
		this.label = label || "Default";
		this.$elem = $( "<button>" ).text( this.label );
	}
	render($where) {
		super.render( $where );
		this.$elem.click( this.onClick.bind( this ) );
	}
	onClick(evt) {
		console.log( "Button '" + this.label + "' clicked!" );
```

```
        }
}
```

Beyond this syntax *looking* nicer, what problems does ES6 solve?

1. There's no more (well, sorta, see below!) references to `.prototype` cluttering the code.
2. `Button` is declared directly to "inherit from" (aka `extends`) `Widget`, instead of needing to use `Object.create(..)` to replace a `.prototype` object that's linked, or having to set with `.__proto__` or `Object.setPrototypeOf(..)`.
3. `super(..)` now gives us a very helpful **relative polymorphism** capability, so that any method at one level of the chain can refer relatively one level up the chain to a method of the same name. This includes a solution to the note from Chapter 4 about the weirdness of constructors not belonging to their class, and so being unrelated -- `super()` works inside constructors exactly as you'd expect.
4. `class` literal syntax has no affordance for specifying properties (only methods). This might seem limiting to some, but it's expected that the vast majority of cases where a property (state) exists elsewhere but the end-chain "instances", this is usually a mistake and surprising (as it's state that's implicitly "shared" among all "instances"). So, one *could* say the `class` syntax is protecting you from mistakes.
5. `extends` lets you extend even built-in object (sub)types, like `Array` or `RegExp`, in a very natural way. Doing so without `class .. extends` has long been an exceedingly complex and frustrating task, one that only the most adept of framework authors have ever been able to accurately tackle. Now, it will be rather trivial!

In all fairness, those are some substantial solutions to many of the most obvious (syntactic) issues and surprises people have with classical prototype-style code.

`class` Gotchas

It's not all bubblegum and roses, though. There are still some deep and profoundly troubling issues with using "classes" as a design pattern in JS.

Firstly, the `class` syntax may convince you a new "class" mechanism exists in JS as of ES6. **Not so.** `class` is, mostly, just syntactic sugar on top of the existing `[[Prototype]]` (delegation!) mechanism.

That means `class` is not actually copying definitions statically at declaration time the way it does in traditional class-oriented languages. If you change/replace a method (on purpose or by accident) on the parent "class", the child "class" and/or instances will still be "affected", in that they didn't get copies at declaration time, they are all still using the live-delegation model based on `[[Prototype]]`:

```
class C {
        constructor() {
                this.num = Math.random();
        }
        rand() {
                console.log( "Random: " + this.num );
        }
}

var c1 = new C();
c1.rand(); // "Random: 0.4324299..."

C.prototype.rand = function() {
        console.log( "Random: " + Math.round( this.num * 1000 ));
};

var c2 = new C();
c2.rand(); // "Random: 867"

c1.rand(); // "Random: 432" -- oops!!!
```

This only seems like reasonable behavior *if you already know* about the delegation nature of things, rather than expecting *copies* from "real classes". So the question to ask yourself is, why are you choosing `class` syntax for something fundamentally different from classes?

Doesn't the ES6 `class` syntax **just make it harder** to see and understand the difference between traditional classes and delegated objects?

`class` syntax *does not* provide a way to declare class member properties (only methods). So if you need to do that to track shared state among instances, then you end up going back to the ugly `.prototype` syntax, like this:

```
class C {
        constructor() {
                // make sure to modify the shared state,
                // not set a shadowed property on the
                // instances!
                C.prototype.count++;

                // here, `this.count` works as expected
                // via delegation
                console.log( "Hello: " + this.count );
        }
}

// add a property for shared state directly to
// prototype object
C.prototype.count = 0;

var c1 = new C();
// Hello: 1

var c2 = new C();
// Hello: 2

c1.count === 2; // true
c1.count === c2.count; // true
```

The biggest problem here is that it betrays the `class` syntax by exposing (leakage!) `.prototype` as an implementation detail.

But, we also still have the surprise gotcha that `this.count++` would implicitly create a separate shadowed `.count` property on both `c1` and `c2` objects, rather than updating the shared state. `class` offers us no consolation from that issue, except (presumably) to imply by lack of syntactic support that you shouldn't be doing that *at all*.

Moreover, accidental shadowing is still a hazard:

```
class C {
        constructor(id) {
                // oops, gotcha, we're shadowing `id()` method
                // with a property value on the instance
                this.id = id;
        }
        id() {
                console.log( "Id: " + this.id );
        }
}

var c1 = new C( "c1" );
c1.id(); // TypeError -- `c1.id` is now the string "c1"
```

There's also some very subtle nuanced issues with how `super` works. You might assume that `super` would be bound in an analogous way to how `this` gets bound (see Chapter 2), which is that `super` would always be bound to one level higher than whatever the current method's position in the `[[Prototype]]` chain is.

However, for performance reasons (`this` binding is already expensive), `super` is not bound dynamically. It's bound sort of "statically", as declaration time. No big deal, right?

Ehh... maybe, maybe not. If you, like most JS devs, start assigning functions around to different objects (which came from `class` definitions), in various different ways, you probably won't be very aware that in all those cases, the `super` mechanism under the covers is having to be re-bound each time.

And depending on what sorts of syntactic approaches you take to these assignments, there may very well be cases where the `super` can't be properly bound (at least, not where you suspect), so you may (at time of writing, TC39 discussion is ongoing on the topic) have to manually bind `super` with `toMethod(..)` (kinda like you have to do `bind(..)` for `this` -- see Chapter 2).

You're used to being able to assign around methods to different objects to *automatically* take advantage of the dynamism of `this` via the *implicit binding* rule (see Chapter 2). But the same will likely not be true with methods that use `super`.

Consider what `super` should do here (against `D` and `E`):

```
class P {
        foo() { console.log( "P.foo" ); }
}

class C extends P {
        foo() {
                super();
        }
}

var c1 = new C();
c1.foo(); // "P.foo"

var D = {
        foo: function() { console.log( "D.foo" ); }
};

var E = {
        foo: C.prototype.foo
};

// Link E to D for delegation
Object.setPrototypeOf( E, D );

E.foo(); // "P.foo"
```

If you were thinking (quite reasonably!) that `super` would be bound dynamically at call-time, you might expect that `super()` would automatically recognize that `E` delegates to `D`, so `E.foo()` using `super()` should call to `D.foo()`.

Not so. For performance pragmatism reasons, `super` is not *late bound* (aka, dynamically bound) like `this` is. Instead it's derived at call-time from `[[HomeObject]].[[Prototype]]`, where `[[HomeObject]]` is statically bound at creation time.

In this particular case, `super()` is still resolving to `P.foo()`, since the method's `[[HomeObject]]` is still `C` and `C.[[Prototype]]` is `P`.

There will *probably* be ways to manually address such gotchas. Using `toMethod(..)` to bind/rebind a method's `[[HomeObject]]` (along with setting the `[[Prototype]]` of that object!) appears to work in this scenario:

```
var D = {
        foo: function() { console.log( "D.foo" ); }
};

// Link E to D for delegation
var E = Object.create( D );

// manually bind `foo`s `[[HomeObject]]` as
// `E`, and `E.[[Prototype]]` is `D`, so thus
// `super()` is `D.foo()`
E.foo = C.prototype.foo.toMethod( E, "foo" );

E.foo(); // "D.foo"
```

Note: `toMethod(..)` clones the method, and takes `homeObject` as its first parameter (which is why we pass `E`), and the second parameter (optionally) sets a `name` for the new method (which keep at "foo").

It remains to be seen if there are other corner case gotchas that devs will run into beyond this scenario. Regardless, you will have to be diligent and stay aware of which places the engine automatically figures out `super` for you, and which places you have to manually take care of it. **Ugh!**

Static > Dynamic?

But the biggest problem of all about ES6 `class` is that all these various gotchas mean `class` sorta opts you into a syntax which seems to imply (like traditional classes) that once you declare a `class`, it's a static definition of a (future instantiated) thing. You completely lose sight of the fact `C` is an object, a concrete thing, which you can directly interact with.

In traditional class-oriented languages, you never adjust the definition of a class later, so the class design pattern doesn't suggest such capabilities. But **one of the most powerful parts** of JS is that it *is* dynamic, and the definition of any object is (unless you make it immutable) a fluid and mutable *thing*.

`class` seems to imply you shouldn't do such things, by forcing you into the uglier `.prototype` syntax to do so, or forcing you think about `super` gotchas, etc. It also offers *very little* support for any of the pitfalls that this dynamism can bring.

In other words, it's as if `class` is telling you: "dynamic is too hard, so it's probably not a good idea. Here's a static-looking syntax, so code your stuff statically."

What a sad commentary on JavaScript: **dynamic is too hard, let's pretend to be (but not actually be!) static**.

These are the reasons why ES6 `class` is masquerading as a nice solution to syntactic headaches, but it's actually muddying the waters further and making things worse for JS and for clear and concise understanding.

Note: If you use the `.bind(..)` utility to make a hard-bound function (see Chapter 2), the function created is not subclassable with ES6 `extend` like normal functions are.

Review (TL;DR)

`class` does a very good job of pretending to fix the problems with the class/inheritance design pattern in JS. But it actually does the opposite: **it hides many of the problems, and introduces other subtle but dangerous ones**.

`class` contributes to the ongoing confusion of "class" in JavaScript which has plagued the language for nearly two decades. In some respects, it asks more questions than it answers, and it feels in totality like a very unnatural fit on top of the elegant simplicity of the `[[Prototype]]` mechanism.

Bottom line: if ES6 `class` makes it harder to robustly leverage `[[Prototype]]`, and hides the most important nature of the JS object mechanism -- **the live delegation links between objects** -- shouldn't we see `class` as creating more troubles than it solves, and just relegate it to an anti-pattern?

I can't really answer that question for you. But I hope this book has fully explored the issue at a deeper level than you've ever gone before, and has given you the information you need *to answer it yourself*.