Prototypes

Every JavaScript object has a prototype. All objects in JavaScript inherit their methods and properties from their prototypes.

Lets check out an example. Open up your Chrome Developer Console (Windows: Ctrl + Shift + J)(Mac: Cmd + Option + J) and type the Student` function from earlier in this article:

```
function Student(name, age) {
  this.name = name;
  this.age = age;
}
```

To prove that every object has a prototype, lets now type:

```
Student.prototype;
// Object {...}
```

Cool, an object is returned. Now lets try creating a new student:

```
var second = new Student('Jeff', 50);
```

We've used our `Student` constructor function to create our second student named Jeff. And, since we used the `new` operator, the `__proto__` property should have also been added to our `second` object. It should point to the parent constructor. Lets try seeing if they're equal:

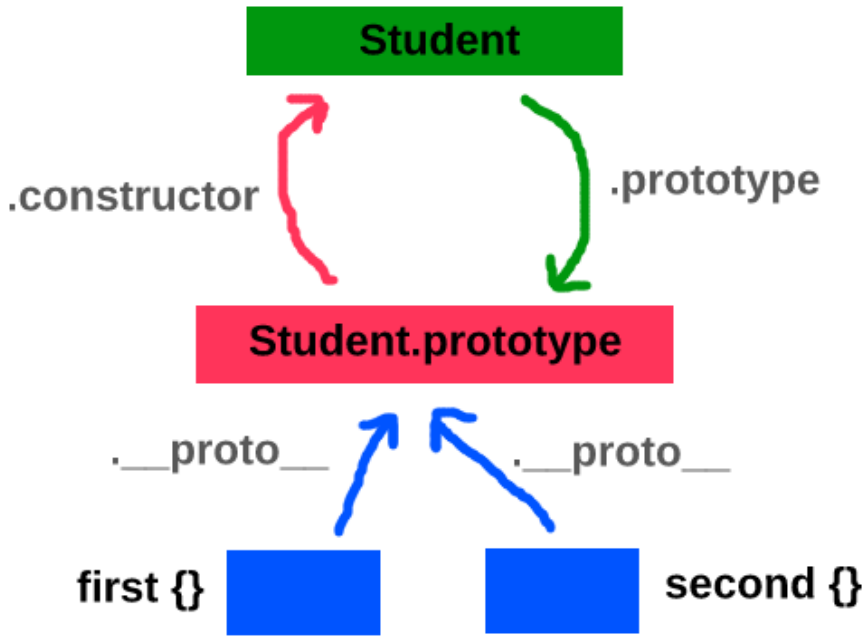
```
second.__proto__ === Student.prototype;
// true
```

Finally, our `Student.prototype.constructor` should point to our `Student` constructor function:

```
Student.prototype.constructor;

//  function Student(name, age) {
//    this.name = name;
//    this.age = age;
//  }
```

This is getting complicated very quickly. Lets see if a crappy *paint* image doesn't help you visualize what is going on:



As you can see above, our `Student` constructor function (as well as all other constructor functions) have a property called `.prototype`. This prototype has an object on it called `.constructor` which points back to the constructor function. It's a nice little loop. Then, when we use the `new` operator to create a new object, each object has `.__proto__` property which links the new object back to the `Student.prototype`.

So why is this so important?

> ***It's important because of inheritance. The prototype object is shared among all objects created with that constructor function. This means we can add functions and properties to the prototype that all of our objects can use.***

In our above examples, we only created two `Student` objects, but what if instead of two students, we have 20,000? All of a sudden, we're saving a ton of processing power by putting shared functions on the prototype instead of in each of the student objects.

Lets look at an example to drive this idea home. In your console add the following line:

```
Student.prototype.sayInfo = function(){
  console.log(this.name + ' is ' + this.age + ' years old');
}
```

Again, what we're doing here is adding a function to the Student prototype — Any student we create or have created now has access to this brand new `.sayInfo` function! Let's test it out:

```
second.sayInfo();
// Jeff is 50 years old
```

Add a new student and try it again:

```
var third = new Student('Tracy', 15);

// Now if we log third out, we see the object only has two
// properties, age and name. Yet, we still have access to the
// sayInfo function:

third;
// Student {name: "Tracy", age: 15}

third.sayInfo();
// Tracy is 15 years old
```

It works! And it works because of *inheritance*. With JavaScript objects, an object will first look to see if it has the property we are calling. If it doesn't, it then moves upwards, to it's prototype and says '*Hey, do you have this property?*'. This same pattern continues all the way up until we either find that property, or we reach the end of the prototype chain at the global object.

Inheritance is the same reason you've been able to use functions like `.toString()` in the past! Think about it, you've never written a `toString()` method, yet you've been able to use it just fine. That's because the method, as well as other built in JS methods are on the `Object prototype`. Every object we create ultimately delegates to the `Object prototype`. And sure, we could over write these methods with something like this:

```
var name = {
  toString: function(){
    console.log('Not a good idea');
  }
};

name.toString();
// Not a good idea
```

Our object first checks to see if it has the method before moving to the prototype. Since we do have the method, it is run and there is no inheritance needed. But that's not a good idea. Leave global methods as they are and name your functions something else.

Conclusion

As a new developer this may be a very tough concept to wrap your head around, but once you do, you can write much better, dryer code. With prototypes we can share the same pieces of code across hundreds, even thousands of objects quickly and effectively. As with all my articles, the idea of this article is to get you a basis of knowledge for you to go out and continue learning more on your own.