Object Techniques

I'd like to show you some of the techniques that are possible using objects, as well. Some of the issues we've discussed about iterating through the items in an object have to do with whether or not the properties are unique to that object.

ECMAScript 5 introduced the `keys` method on the base object and this lets you get an object own enumerable properties.

```
var fruits = {};
fruits.apple = "red";
fruits.orange = "orange";
fruits.pear = "green";
console.log(Object.keys(fruits)); //["apple", "orange", "pear"]
```

We get an array that contains just the keys from our object. Now let's add `extender` once again:

```
Object.prototype.extender = "Hahaha";
console.log(Object.keys(fruits)); //["apple", "orange", "pear"]

for (var item in fruits) {
  console.log("fruits." + item + " = " + fruits[item]);
}
//"fruits.apple = red"
//"fruits.orange = orange"
//"fruits.pear = green"
//"fruits.extender = Hahaha"
```

Notice that keys remain the same, however when we output all items in a `for` loop, `extender` is being printed out as well.

New Object Types

You can use the `this` keyword to make properties and methods public. This can be useful for creating object types that have private variables and public variables, some of which are shared across all objects and some of which are only accessible within the scope of that one object.

```
var Food = function(name) {
  var label = name;
  this.edible = true;
  this.formal = function() {
    return name.toUpperCase() + " ESQ.";
  };
};

var apple = new Food("apple");
console.log(apple.label); //unedined
console.log(apple.edible); //true
console.log(typeof apple); //"object"
console.log(apple.formal()); //"APPLE, ESQ."
```

So `label` is a private variable - we can't reference it from outside. `edible` and `formal`, on the other hand, are public and we can reference them without any problems.

Assigning Methods

Methods that are redundant and live on each instance created from the same constructor take up extra space in memory. But methods on new types can also be assigned to a prototype, and that way they're shared across all of the instances. And that can save resources. Methods that are defined externally on the prototype, however, only have access to the public properties of the instance. That means that you can only access properties inside of each instance that have been defined with the `this` keyword.

```
var Food = function(name) {
//var label = name;
  this.label = name;
  this.edible = true;
};

// Food.prototype.formal = function() {
//   return label.toUpperCase() + ", ESQ.";
// };

Food.prototype.formal = function() {
  return this.label.toUpperCase() + ", ESQ.";
};

var apple = new Food("apple");
console.log(apple.label); //"apple"
console.log(apple.edible); //true
console.log(typeof apple); //"object"
console.log(apple.formal()); //"APPLE, ESQ."
```

We can't reference `label` directly, as we are outside of the constructor and `label` is not defined outside of this constructor. That's why we have to use `this.label` instead.

ECMAScript 6 Classes

The syntax for this looks a little bit messy. That's one of the reasons why in ECMAScript 6 JavaScript introduced the concept of the class syntax for defining object types. The syntax doesn't actually change the limitations of prototypal inheritance, but it creates a pretty wrapper around it. So, that we can write classes that behave the way that we expect them to. It's basically the same prototype inheritance syntax that wehad in JavaScript from ECMA Script Five and earlier, but just wrapped in some pretty syntactic sugar, to make it easier to swallow.