Object Creation and Prototypes

Notes

There are three main ways to create objects in Javascript. We mention them now but will discuss their particulars a bit later:

Object literal What we've been using: `{ foo: bar, foo2: baz }`
Introduced in ES5.

Object Constructors Using `new`. So something like `new Object()`, `new Array()` etc. We will discuss these more later.

For now we focus on `Object.create`, as it offers the most direct way to discuss prototypes.

Prototypes

Every object has another object called its **prototype**.

The object *inherits* properties of its prototype.

A simple way to set an object's prototype, is via `Object.create` as the first argument.

Here is a simple example of that:

```
var pete = { name: "Pete", sayHi: function() { return "Hi! I am " + this.name; } };
pete.sayHi();
var myka = Object.create(pete);
myka.name = "Myka";
myka.sayHi();
```

Notice the use of `this` to refer to the object currently calling the function.

Let's think about what happens here:

- The `myka` object is asked to find and call a function `sayHi`.
- It only has a property called `name`, so it asks its "prototype", `pete`, to see if *it* has a `sayHi` property.
- `pete` says that it does (if it did not, it would delegate the question to its prototype and so on and so forth). It returns its `sayHi` function, and `myka` invokes it.
- This is a method invocation, so `this` is set to `myka`. Therefore when that functions looks for `this.name`, it finds `"Myka"`.

There is a secret property of all objects, called `__proto__` (that's two underscores on each side), that points to this prototype object. It is generally advisable not to mess with that property. So we could do:

```
myka.__proto__
myka.__proto__ === pete
```

Object methods and properties are looked up the prototype chain, which eventuallly ends in the null object.

Prototypes are a good place to put shared methods and values, that all objects of a certain "type/class" should have access to. With the use of this, these functions can have access to the object that they are meant to represent. We will see examples of this in the stack examples of the next segment.

Constructors and Prototypes

Constructors are special functions that are meant to be used for creating objects. They are meant to always be called with the keyword new, and by convention are always capitalized. You have already seen a number of these constructors, but did not know they were that:

- Object[2]
- Number[3]
- Array[4]
- Function[5]
- RegExp[6]
- ADD YOUR CLASS HERE!

All of these are actually "constructors". You can think of them a little bit like the analog of a class in other languages. And you can create your own: All you need is a function, with a couple of considerations:

- You should capitalize your function's name as reminder to those calling it that they must prepend it with new.
- In the body of the function, this will refer to a newly created object.
- The function will implicitly return this, unless you return another object instead.
- The new object's __proto__ will be set to the prototype property of your function (functions are regular objects, you can add properties to them).

```
function Foo(v) {
    console.log("Proto set to: ", this.__proto__);
    this.v = v;
}
Foo.prototype = {
    bar: "I am a Foo"
};
var a = new Foo(5);
```

[2] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object
[3] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number
[4] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array
[5] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function
[6] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp

```
a.v;
a.bar;
Foo.prototype.baz = "Method added after creation, woohoo!";
a.baz;
var b = new Foo(10);
a.baz = "What did I change?";
b.baz;
```