Understanding "Prototypes" in JavaScript

• 12 Aug 2011

For the purposes of this post, I will be talking about JavaScript objects using syntax defined in ECMAScript 5.1. The basic semantics existed in Edition 3, but they were not well exposed.

A Whole New Object

In JavaScript, objects are pairs of keys and values (in Ruby, this structure is called a Hash; in Python, it's called a dictionary). For example, if I wanted to describe my name, I could have an object with two keys: `firstName` would point to "Yehuda" and `lastName` would point to "Katz". Keys in a JavaScript object are Strings.

To create the simplest new object in JavaScript, you can use `Object.create`:

```
var person = Object.create(null); // this creates an empty objects
```

Why didn't we just use `var person = {};`? Stick with me! To look up a value in the object by key, use bracket notation. If there is no value for the key in question, JavaScript will return `undefined`.

```
person['name'] // undefined
```

If the String is a valid identifier[1], you can use the dot form:

```
person.name // undefined
```

[1] in general, an identifier starts with a unicode letter, $, _, followed by any of the starting characters or numbers.

A valid identifier must also not be a reserved word. There are other allowed characters, such as unicode combining marks, unicode connecting punctuation, and unicode escape sequences. Check out the spec for the full details

Adding values

So now you have an empty object. Not that useful, eh? Before we can add some properties, we need to understand what a property (what the spec calls a "named data property") looks like in JavaScript.

Obviously, a property has a name and a value. In addition, a property can be **enumerable**, **configurable** and **writable**. If a value is enumerable, it will show up when enumerating over an object using a `for(prop in obj)` loop. If a property is writable, you can replace it. If a property is configurable, you can delete it or change its other attributes.

In general, when we create a new property, we will want it to be enumerable, configurable, and writable. In fact, prior to ECMAScript 5, that was the only kind of property a user could create directly.

We can add a property to an object using `Object.defineProperty`. Let's add a first name and last name to our empty object:

```
var person = Object.create(null);
Object.defineProperty(person, 'firstName', {
  value: "Yehuda",
  writable: true,
  enumerable: true,
  configurable: true
});

Object.defineProperty(person, 'lastName', {
  value: "Katz",
  writable: true,
  enumerable: true,
  configurable: true
});
```

Obviously, this is extremely verbose. We can make it a bit less verbose by eliminating the

common defaults:

```
var config = {
  writable: true,
  enumerable: true,
  configurable: true
};

var defineProperty = function(obj, name, value) {
  config.value = value;
  Object.defineProperty(obj, name, config);
}

var person = Object.create(null);
defineProperty(person, 'firstName', "Yehuda");
defineProperty(person, 'lastName',   "Katz");
```

Still, this is pretty ugly to create a simple property list. Before we can get to a prettier solution, we will need to add another weapon to our JavaScript object arsenal.

Prototypes

So far, we've talked about objects as simple pairs of keys and values. In fact, JavaScript objects also have one additional attribute: a pointer to *another* object. We call this pointer the object's *prototype*. If you try to look up a key on an object and it is not found, JavaScript will look for it in the prototype. It will follow the "prototype chain" until it sees a `null` value. In that case, it returns `undefined`.

You'll recall that we created a new object by invoking `Object.create(null)`. The parameter tells JavaScript what it should set as the Object's *prototype*. You can look up an object's prototype by using `Object.getPrototypeOf`:

```
var man = Object.create(null);
defineProperty(man, 'sex', "male");

var yehuda = Object.create(man);
defineProperty(yehuda, 'firstName', "Yehuda");
defineProperty(yehuda, 'lastName', "Katz");
```

```
yehuda.sex       // "male"
yehuda.firstName // "Yehuda"
yehuda.lastName  // "Katz"

Object.getPrototypeOf(yehuda) // returns the man object
```

We can also add functions that we share across many objects this way:

```
var person = Object.create(null);
defineProperty(person, 'fullName', function() {
  return this.firstName + ' ' + this.lastName;
});

// this time, let's make man's prototype person, so all
// men share the fullName function
var man = Object.create(person);
defineProperty(man, 'sex', "male");

var yehuda = Object.create(man);
defineProperty(yehuda, 'firstName', "Yehuda");
defineProperty(yehuda, 'lastName', "Katz");

yehuda.sex        // "male"
yehuda.fullName() // "Yehuda Katz"
```

Setting Properties

Since creating a new writable, configurable, enumerable property is pretty common, JavaScript makes it easy to do so using assignment syntax. Let's update the previous example using assignment instead of `defineProperty`:

```
var person = Object.create(null);

// instead of using defineProperty and specifying writable,
// configurable, and enumerable, we can just assign the
// value directly and JavaScript will take care of the rest
person['fullName'] = function() {
```

```
  return this.firstName + ' ' + this.lastName;
};

// this time, let's make man's prototype person, so all
// men share the fullName function
var man = Object.create(person);
man['sex'] = "male";

var yehuda = Object.create(man);
yehuda['firstName'] = "Yehuda";
yehuda['lastName'] = "Katz";

yehuda.sex        // "male"
yehuda.fullName() // "Yehuda Katz"
```

Just like when looking up properties, if the property you are defining is an *identifier*, you can use dot syntax instead of bracket syntax. For instance, you could say `man.sex = "male"` in the example above.

Object Literals

Still, having to set a number of properties every time can get annoying. JavaScript provides a literal syntax for creating an object and assigning properties to it at one time.

```
var person = { firstName: "Paul", lastName: "Irish" }
```

This syntax is approximately sugar for:

```
var person = Object.create(Object.prototype);
person.firstName = "Paul";
person.lastName  = "Irish";
```

The most important thing about the expanded form is that object literals *always* set the newly created object's prototype to an object located at `Object.prototype`. Internally, the object literal looks like this:

prototype-chain.png

The default `Object.prototype` dictionary comes with a number of the methods we have come to expect objects to contain, and through the magic of the prototype chain, all new objects created as object literal will contain these properties. Of course, objects can happily override them by defining the properties directly. Most commonly, developers will override the `toString` method:

```
var alex = { firstName: "Alex", lastName: "Russell" };

alex.toString() // "[object Object]"

var brendan = {
  firstName: "Brendan",
  lastName: "Eich",
  toString: function() { return "Brendan Eich"; }
};

brendan.toString() // "Brendan Eich"
```

This is especially useful because a number of internal coercion operations use a supplied `toString` method.

Unfortunately, this literal syntax only works if we are willing to make the new object's prototype `Object.prototype`. This eliminates the benefits we saw earlier of sharing properties using the prototype. In many cases, the convenience of the simple object literal outweighs this loss. In other cases, you will want a simple way to create a new object with a particular prototype. I'll explain it right afterward:

```
var fromPrototype = function(prototype, object) {
  var newObject = Object.create(prototype);

  for (var prop in object) {
    if (object.hasOwnProperty(prop)) {
      newObject[prop] = object[prop];
    }
  }

  return newObject;
};

var person = {
  toString: function() {
    return this.firstName + ' ' + this.lastName;
  }
};

var man = fromPrototype(person, {
  sex: "male"
});

var jeremy = fromPrototype(man, {
  firstName: "Jeremy",
  lastName:  "Ashkenas"
});

jeremy.sex        // "male"
jeremy.toString() // "Jeremy Ashkenas"
```

Let's deconstruct the `fromPrototype` method. The goal of this method is to create a new object with a set of properties, but with a particular prototype. First, we will use `Object.create()` to create a new empty object, and assign the prototype we specify. Next, we will enumerate all of the properties in the object that we supplied, and copy them over to the new object.

Remember that when you create an object literal, like the ones we are passing in to `fromPrototype`, it will always have `Object.prototype` as its prototype. By default, the properties that JavaScript includes on `Object.prototype` are *not enumerable*, so we don't have to worry about `valueOf` showing up in our loop. However, since `Object.prototype` is an object like any other object, anyone can define a new property on it, which may (and probably would) be marked enumerable.

As a result, while we are looping through the properties on the object we passed in, we want to restrict our copying to properties that were defined on the object itself, and not found on the prototype. JavaScript includes a method called `hasOwnProperty` on `Object.prototype` to check whether a property was defined on the object itself. Since object literals will always have `Object.prototype` as their prototype, you can use it in this manner.

The object we created in the example above looks like this:

prototype-chain-2.png

Native Object Orientation

At this point, it should be obvious that prototypes can be used to inherit functionality, much like traditional object oriented languages. To facilitate using it in this manner, JavaScript provides a `new` operator.

In order to facilitate object oriented programming, JavaScript allows you to use a Function object as a combination of a prototype to use for the new object and a constructor function to invoke:

```
var Person = function(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

Person.prototype = {
  toString: function() { return this.firstName + ' ' + this.lastName; }
```

```
}
```

Here, we have a single Function object that is both a constructor function and an object to use as the prototype of new objects. Let's implement a function that will create new instances from this `Person` object:

```
function newObject(func) {
  // get an Array of all the arguments except the first one
  var args = Array.prototype.slice.call(arguments, 1);

  // create a new object with its prototype assigned to func.prototype
  var object = Object.create(func.prototype);

  // invoke the constructor, passing the new object as 'this'
  // and the rest of the arguments as the arguments
  func.apply(object, args);

  // return the new object
  return object;
}

var brendan = newObject(Person, "Brendan", "Eich");
brendan.toString() // "Brendan Eich"
```

The `new` operator in JavaScript essentially does this work, providing a syntax familiar to those comfortable with traditional object oriented languages:

```
var mark = new Person("Mark", "Miller");
mark.toString() // "Mark Miller"
```

In essence, a JavaScript "class" is just a Function object that serves as a constructor plus an attached prototype object. I mentioned before that earlier versions of JavaScript did not have `Object.create`. Since it is so useful, people often created something like it using the `new` operator:

```
var createObject = function (o) {
  // we only want the prototype part of the `new`
```

```
  // behavior, so make an empty constructor
  function F() {}

  // set the function's `prototype` property to the
  // object that we want the new object's prototype
  // to be.
  F.prototype = o;

  // use the `new` operator. We will get a new
  // object whose prototype is o, and we will
  // invoke the empty function, which does nothing.
  return new F();
};
```

I really love that ECMAScript 5 and newer versions have begun to expose internal aspects of the implementation, like allowing you to directly define non-enumerable properties or define objects directly using the prototype chain.

WRITTEN BY

Yehuda Katz

NEXT POST

New Hope for The Ruby Specification

For a few years, a group of Japanese academics have been working on formalizing the Ruby programming language into...

05 SEP 2011

PREVIOUS POST

Understanding JavaScript Function Invocation and "this"

Over the years, I've seen a lot of confusion about JavaScript function invocation. In particular, a lot of people...

11 AUG 2011

Share this post

All content copyright **Katz Got Your Tongue** © 2016 • All rights reserved.
Theme created by **golem.io** • Proudly published with **Ghost**